# Enabling and Improving Centralized Control in Network and Cyber-Physical Systems: An Application-Driven Approach

by

Yikai Lin

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2020

Doctoral Committee:

Professor Z. Morley Mao, Chair
Associate Professor Kira Barton
Assistant Professor N M Mosharaf Kabir Chowdhury
Associate Professor Harsha Madhyastha

Yikai Lin

yklin@umich.edu

ORCID iD: 0000-0002-7637-002X

*To my love, my family, my friends,*

*and in memory of Professor Jun Bi.*

# ACKNOWLEDGEMENTS

After 20 years in school, the journey is finally coming to an end, leaving behind many fond memories. Fortunately, I was never alone in these memories, for so many shared this journey with me. Their company, support, and wisdom helped me reach the highs and through the lows. Words can hardly describe my deepest gratitude to every one of them.

To my advisor, Professor Zhuoqing Morley Mao, thank you for giving me this opportunity and your continuous support and guidance for the past five years. Your keen insights gave me great inspirations in tackling the most challenging problems. Your constructive feedback helped me become a better researcher, engineer, presenter, *etc*. I am forever grateful for this amazing Ph.D. experience with you, and I will truly miss your critical comments as well as encouragement.

To my dissertation committee member and long-time collaborator, Professor Kira Barton, thank you for your kind support for the past four years. Working with you was a joy. More importantly, our collaboration across Computer Science and Mechanical Engineering broadened my horizon and contributed immensely to my dissertation research.

To my other dissertation committee members, Professor Mosharaf Chowdhury and Professor Harsha Madhyastha, thank you for your time and insightful feedback on improving my dissertation work. I learned a great deal from both your classes and your publications. Your works in networked systems deepened my understanding of this field.

To my internship mentors and industry collaborators, Dr. Ulaş Kozat, John Kaippallimalil, and Dr. Anthony C.K. Soong at Huawei R&D, Dr. Ajay Mahimkar, Dr. Bo Han, Dr. Zihui Ge, and Dr. Vijay Gopalakrishnan at AT&T Labs Research, Dr. Yi Wang, Ryan

Zuklie, Dr. Qianwen Yin, and Dr. Keqiang He at Google, thank you for your tutoring and support during and after my internships. You gave me opportunities to work on the most exciting challenges in production networks, some of which became part of this dissertation. You also taught me research and engineering skills that benefited me as a Ph.D. student and will continue to benefit me in my future career.

To my other student collaborators, Ruowang Zhang, Jianbin Zhang, and Junpeng Guo, thank you for your hard work and contribution to my dissertation research. I wish you the best of luck in your future endeavors.

To my roommate, Dr. Zhiqiang Sui, my friends from the RobustNet Research Group, Dr. Mehrdad Moradi, Dr. Yihua Guo, Professor Qi Alfred Chen, Dr. Sanae Rosen, Dr. Ashkan Nikravesh, Dr. Yunhan Jia, Dr. Yuru Shao, Dr. Ke Hong, Dr. Shichang Xu, Chao Kong, Jeremy Erickson, Xiao Zhu, Jie You, Yulong Cao, Shengtuo Hu, Won Park, Jiachen Sun, Xumiao Zhang, Eric Newberry, Jiwon Joung, Can Carlak, Qingzhao Zhang, Dr. Ying Zhang, and Professor Feng Qian, and from the Department of Mechanical Engineering at UM, Professor Dawn Tilbury, Dr. James Moyne, Efe Balta, Dr. Ilya Kovalenko, Dr. Yassine Qamsane, Dr. Felipe Lopez, Dr. Miguel Saez, Dr. Zheng Wang, thank you for all the love, help and support. It was a pleasure knowing all of you, and I will miss you.

To the CSE staff at UM, who made our doctoral study as smooth as it could be, Dawn Freysinger, Ashley Andreae, Stephen Reger, and Karen Liska, among others, thank you for all your time and support.

To my father, Weibin Lin, my mother, Rongyuan Luo, my grandma, Sumei Luo, and my aunt, Suiyuan Luo, thank you so much for your unconditional love, support and sacrifice for all these years. Thank you for always standing behind me when I chase my dreams.

To my lovely wife, Yunyan Zhang, thank you for accepting me as who I am and always trusting me since high school. I was extremely fortunate to have met you, and since then, I could never see myself without you. I feel as excited as ever about spending the rest of my life with you. To quote Dr. Leonard Hofstadter from one of my favorite sitcoms *The Big*

*Bang Theory*: “Our babies will be smart and beautiful.”

I want to dedicate this dissertation to my undergrad research advisor at Tsinghua University, Professor Jun Bi, for trusting me and providing me with all the tutoring and support I could have asked for. He was the reason I decided to and was able to pursue a Ph.D. He devoted all his life to his students and this community, but he left us too early. I miss him with all my heart, and I wish him nothing but peace.

# TABLE OF CONTENTS

# LIST OF FIGURES

**Figure**

# LIST OF TABLES

**Table**

# LIST OF ABBREVIATIONS

**CPS** Cyber-Physical Systems

**DT** Digital Twin

**IoT** Internet-of-Things

**MOPs** Methods of Procedure

**NFV** Network Function Virtualization

**PPE** Personal Protective Equipment

**PB** Programmable Buffer

**QoS** Quality of Service

**SDN** Software-Defined Networks

**WAN** Wide-Area Networks

# ABSTRACT

Cloud providers and carriers are actively adopting a centralized control paradigm, as in Software-Defined Networks (SDN), to achieve high flexibility and agility in network management. This paradigm allows applications (apps) to easily monitor/reconfigure network devices based on a global view. With emerging hardware (*e.g.,* internet-of-things, autonomous/connected vehicles, smart manufacturing) and software technologies (*e.g.,* digital twins), Cyber-Physical Systems (CPS) can also leverage this paradigm to improve their performance and reliability. However, to reflect rapidly emerging device capabilities and use cases, apps need to evolve constantly, which introduces new requirements for programmability, extensibility, and data availability. As a result, it is challenging and sometimes impossible to deploy existing SDN(-like) solutions as they are without resorting to ad hoc patches that are time-consuming to develop and hardly reusable.

In this dissertation, we argue that (1) it is possible to systematically address such problems without tailoring to specific apps, and (2) generalization of apps' behaviors is key to the solutions. To support these claims, we present three systematic solutions that enable and improve centralized control in different network systems and CPS. For apps that buffer network packets during device mobility to prevent loss and reordering, we expand SDN programmability to support in-network buffering with Programmable Buffer. For apps running in a heterogeneous network, we generalize SDN traffic management abstractions with Egret. To enable centralized control in CPS and support data-driven apps, we develop an extensible and generic framework named SDNator. We demonstrate that these solutions can not only support legacy and emerging apps but facilitate the innovation of new ones.

# CHAPTER I

# Introduction

## 1.1 Background and Thesis Statement

More than ten years after the introduction of the OpenFlow protocol [1], SDN-based and -inspired solutions have seen widespread deployment in data centers [2, 3], Wide-Area Networks (WAN) [4, 5, 6], and cellular networks [7, 8]. The SDN paradigm profoundly influenced our view of network management [9, 10, 11, 4, 5, 12] and network hardware designs [13, 14, 15, 16, 17]. This paradigm, in essence, describes *a centralized control architecture where applications (the S in SDN) possess the intelligence of the system and fulfill many roles such as computing, decision making, and reconfiguration (of devices) while leveraging the global view provided by a (logically) centralized controller* (Figure 1.1). Compared to traditional distributed approaches, centralized solutions substantially improve the flexibility and efficiency of device management, simplify and speed up software development and iteration, and open up exciting opportunities for new applications/services/strategies [18, 19, 20] that would be otherwise infeasible in a traditional distributed approach.

What makes this paradigm even more appealing is its broad applicability beyond network systems. For example, its capability to aggregate data from different devices is crucial for emerging CPS applications, especially Digital Twin (DT). DTs are software replicas of physical devices that keep track of real-time information such as status, states, and phys-

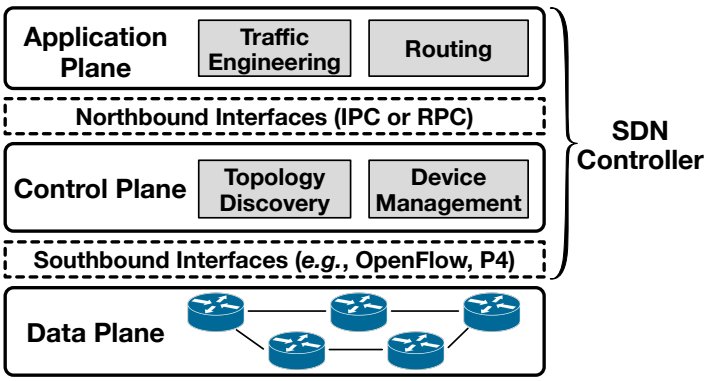



Figure 1.1: Software-Defined Networks (SDN)

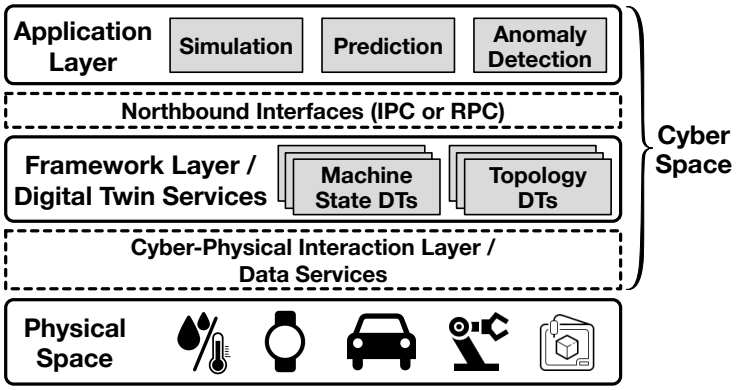



Figure 1.2: Digital-Twin-equipped Cyber-Physical Systems (CPS)

ical properties. This information provides additional monitoring and analysis capabilities to other applications (Figure 1.2), which helps improve system performance and reliability. Therefore, DT is considered one of the key enablers for visions like smart manufacturing and Industry 4.0 [21, 22]. With increasingly capable hardware such as the Internet-of-Things (IoT), autonomous/connected vehicles, and 3D printers, an SDN-like paradigm is both desirable and feasible in CPS. Many recent studies have demonstrated that centralized control paradigms in various CPS can achieve higher efficiency, performance, security, and reliability compared to decentralized approaches [23, 24, 25, 26, 27, 28].

To realize this centralized paradigm, there needs to be systems that provide the proper interfaces, abstractions and data to the applications, as shown in Figure 1.1 and 1.2. In SDN, there is an abundance of these "controller" systems that support applications using the OpenFlow protocol (*e.g.,* NOX [9], Ryu [29] and Floodlight [30]) and/or the P4 [31]

specifications (*e.g.,* P4Runtime [32], OpenDaylight [33], and ONOS [34]). However, with new services (*e.g.,* 5G [18]), increasingly diverse network compositions [35], and highly heterogeneous CPS [36], it is extremely difficult to directly use these existing systems as they fall short of the new programmability, generality and data availability requirements.

- **Limited programmability.** Many critical applications in current LTE networks, such as Mobility Management and Paging, need to buffer packets during user mobility in order to prevent loss and preserve ordering. In 5G networks, these buffering behaviors will be even more essential with new applications such as machine-to-machine communications. Yet today's SDN programming abstractions, be they OpenFlow-based or P4-based, revolve around parsing, modification, and routing of packets. Existing SDN systems cannot provide the necessary interfaces and abstractions to support applications' packet buffering behaviors.

- **Limited generality.** Traffic migration is one of the most common network operations. Existing SDN systems use switches and routers to perform traffic migration, which only suits certain data center networks with homogeneous network compositions. In cellular core networks, however, the data-plane consists of many different types of devices and functions, which makes traffic management extremely complex. Network Function Virtualization (NFV) will add to this complexity by allowing even more network functions to be developed and deployed much easier.

- **Limited data availability and extensibility.** Existing SDN controllers commonly adopt an event-driven model that minimizes southbound communications and control-plane overhead. This model satisfies most existing network applications but falls short of meeting the requirements of future network applications and most CPS applications which are predominantly data-driven. Moreover, existing SDN controllers are highly tailored to network applications and lack the extensibility to accommodate vastly different CPS applications and data types.

Table 1.1: Summary of dissertation work

| **Problem Scope** | **App Behaviors** | **Project** |
|---|---|---|
| Expand SDN programmability | Buffer traffic during device mobility | Pausing and Resuming Network Flows Using Programmable Buffers |
| Improve SDN generality | Migrate traffic to or from a device | Egret: Simplifying Traffic Management for Physical and Virtual Network Functions |
| New system architecture for data-driven apps | Produce & consume a lot of real time and historical data | ADD: Application and Data-Driven Controller Design |
| | | SDNator: Enabling Extensible and Data-Driven Control in Cyber-Physical Systems |

These gaps between new application requirements and limited system capabilities make it particularly challenging to deploy existing SDN-based solutions as they are in many network and CPS without significant performance or usability penalties. For example, in cellular networks, without a data-plane packet buffering functionality and corresponding programming abstractions, applications will have to buffer packets on the control-plane, which suffers packet reordering and loss due to high delays and low bandwidth of the control path [18]. Adapting/modifying existing systems based on specific application requirements is infeasible in the long term, especially in CPS which have unparalleled heterogeneity in devices, data, protocols, and applications. If every new device/application requires a new programming abstraction or interface, the value of having a flexible and agile centralized control paradigm diminishes.

Rather than applying ad hoc patches to existing SDN solutions that are time-consuming to develop and barely reusable, in this dissertation, we demonstrate that it is possible to systematically address these limitations using an application-driven approach. We comprehensively study the characteristics and behaviors of applications of interest, and generalize these behaviors into abstracted requirements to help guide the designs of our solutions. We describe three general systematic solutions in this dissertation that enable and improve centralized control in different networks and CPS. Table 1.1 summarizes the four works from three different perspectives that support the thesis statement:

**Thesis Statement**. Generalization of application behaviors, combined with techniques such as abstraction and modularization, is key to improving system programmability, generality, data availability, and extensibility to enable centralized control in network and cyber-physical systems.

## 1.2 Overview

To the best of our knowledge, we are the first to systematically study the limitations of existing SDN solutions in programmability, generality, and data-availability, to support emerging applications across network and cyber-physical systems. We (1) carry out case studies to extract and generalize requirements and behaviors of different applications, (2) analyze and compare existing SDN solutions' capabilities and limitations, and (3) design general programming abstractions, interfaces, and architecture to effectively address the identified limitations. We show that our solutions can satisfy the requirements of legacy and emerging applications while enabling the development of new applications.

This dissertation is organized as follows. As shown in Table 1.1, we put the four works under three sub-topics: "expand SDN programmability" (Chapter II), "improve SDN generality" (Chapter III), and "new system architecture for data-driven apps" (Chapter IV and V).

In Chapter II, we present "Pausing and Resuming Network Flows Using Programmable Buffers" [18], where we enable centralized control of in-network buffering by expanding existing SDN programming abstractions. Specifically, we design (1) a new yet fully backward-compatible programming abstraction called Programmable Buffer (PB) for network buffering, (2) a set of northbound APIs for network applications to pause and resume traffic flows conveniently in the network, (3) a set of southbound APIs to efficiently manage buffer states and buffering operations, and (4) a new mobility management application for high-frequency handovers in 5G. We implement PB using Docker containers [37], soft-

ware switches [38] and Ryu SDN controller [29]. Our benchmarks show that PB exceeds 5G standards in every aspect tested. It can deliver over 90 Gbps throughput with large packets, and scale out on programmable switches with acceptable overhead. Moreover, the PB abstraction is powerful enough to handle the 5G extreme mobility scenarios with less than 5% performance drop.

In Chapter III, we present "Egret: Simplifying Traffic Management for Physical and Virtual Network Functions" [35], where we improve centralized control of traffic migration in heterogeneous data-planes. We perform an extensive analysis of over 200 Methods of Procedure (MOPs) from a major U.S. carrier, which suggests that generalizing traffic migration with a unified model is feasible. Based on the findings, we design Egret, a generic traffic migration system that simplifies traffic management for physical and virtual network functions. Egret (1) hides intricate implementation details from operators with generic intention-based interfaces, and (2) modularizes common traffic migration procedures to enable plug-and-play by developers and vendors. Leveraging a new mask-based abstraction of traffic migration jobs, Egret can further simplify reverse traffic migration and enable job interleaving. We implement Egret and integrate it with three real-world network functions to qualitatively evaluate its generality and simplicity. For some procedures, Egret can reduce operational complexity by over 99%.

In Chapter IV, we present "ADD: Application and Data-Driven Controller Design" [36], where we study the unique characteristics and requirements of data-driven applications and identify the gaps between these requirements and designs of existing SDN controllers. In particular, we carry out a case study on smart manufacturing systems, which have highly heterogeneous device compositions, and applications that are much less "throughput" hungry or "latency" sensitive than network applications but require a lot more data for (real-time) decision making. We share the insights we gain that help us design a new Application and Data-Driven (ADD) model for SDN controllers. We build a proof-of-concept ADD controller based on this model and develop two applications to showcase

its new capabilities. Evaluation results show that ADD delivers satisfying scalability and performance. More importantly, applications enabled by ADD gain more insights into the data plane and can make better decisions faster.

In Chapter V, we present "SDNator: Enabling Extensible and Data-Driven Control in Cyber-Physical Systems", where we incorporate the insights from Chapter IV and develop the first framework for building centralized controllers in CPS. SDNator achieves extensibility through plug-and-play of applications with no controller adaptations or modifications. It achieves generality through its data-driven abstractions, allowing it to integrate with different CPS applications and systems easily. SDNator supports both event-driven and data-driven programming patterns and delivers over 100k messages/s while incurring less than 100us delay. We demonstrate how easy it is to enable centralized control using SDNator (either through developing new CPS applications or integrating existing ones) in our case studies of network and additive manufacturing systems. Most notably, we carry out the first study on digital-twin-equipped centralized control of additive manufacturing fleets and show that it shortens regular production time by up to nearly 40%, and Personal Protective Equipment (PPE) production time by more than 50% compared to a baseline distributed approach.

In Chapter VI, we summarize the key contributions of this dissertation, discuss limitations of our approach, and describe potential future research directions.

# CHAPTER II

# Pausing and Resuming Network Flows Using Programmable Buffers

In this chapter, we focus on applications' programmability requirements. In particular, we look at applications that buffer packets in the network to prevent packet loss or reordering during communication interruptions such as device mobility. We discuss how existing SDN systems lack the programmability to efficiently support these applications, which are common and critical in cellular networks, especially in 5G. We develop a new data-plane component, Programmable Buffer (PB), and expand the SDN programming abstractions to allow applications to efficiently control where, when, and how a network flow is buffered. PB is backward-compatible with existing SDN applications and delivers performance that exceeds 5G standards. More importantly, PB enables us to develop a new mobility management application for extreme handover scenarios in 5G networks.

## 2.1 Introduction

As mobile networks transition toward 5G, there is an opportunity to fundamentally re-architect the core network to achieve scalability, flexibility and service agility [39, 40]. These features become especially crucial as more types of devices (IoT, virtual reality, autonomous vehicles) are joining 5G. NFV and SDN are considered the key enablers of these

features [41]. NFV replaces hardware boxes with software instances running in VMs or containers, and allows multiple instances (of one or more network functions) to share the same commodity servers. This simplifies the development, deployment, and management of network functions, and significantly reduces the costs. SDN further simplifies the deployment and management of network services by programmatically meshing up network functions together. In SDN paradigm, logically centralized control applications programmatically change the forwarding and packet processing behavior of distributed data-plane nodes with frameworks like OpenFlow [1] and P4 [31]. These frameworks have significantly pushed the boundary of network programming. However, little advancement is made on a critical behavior of cellular networks: network buffering. Neither OpenFlow nor P4 has control over where, when and how a network flow is buffered inside the network, except for sending the flows to the controller [42].

Why is network buffering critical? Many existing services in cellular networks such as Mobility Management and Paging rely on network buffering to guarantee loss-free and order-preserving delivery during user mobility. While these services will remain fundamental in 5G networks, they will not be considered as fixed functions [43]. Instead, they will be customized for different network slices [44]. Furthermore, new set of mobile edge services that benefit from in-network caching and storage should be supported when and where needed. As a brute-force approach, these services can be deployed as VNFs in central offices. However, with a proper set of abstractions for flow buffering and a set of APIs to manage flow buffering behavior, we can build a network buffering service that can be consumed easily, efficiently, and flexibly by many network services.

Towards this end, we propose Programmable Buffer (PB), a new programming abstraction for managing where, when and how a network flow is buffered within the network. PB incorporates the new programming abstractions into existing SDN programming models and provides both northbound and southbound APIs to efficiently manage flow buffering behaviors on software switches. A PB enabled switch, Programmable Buffer Switch, ex-

poses buffering operations to the control-plane via a set of low-level (southbound) APIs. A PB service running on the SDN controller wraps up these APIs into high-level (northbound) APIs for control applications, greatly simplifying the process of buffer-based programming.

PB's low-level APIs manage the available memory on **software switches** for programmable buffers, and configure them as on-switch traffic sources and sinks. When combined with SDN's existing fine-grained flow control, applications can easily pause flows in the network, store packets for an arbitrary duration and resume/play back flows later towards any path as dictated. These functions allow PB to easily support existing services like mobility management, and enable new applications like fast mobility[1] management and connectionless communications in the 5G era.

To summarize, in this chapter, we make the following contributions:

- We propose Programmable Buffer (PB), a novel SDN-based approach for managing flow buffering in a network. PB abstractions allow core network services to be further decoupled (by keeping buffering functionality on the data-plane), which helps enable a scalable high-performance 5G network.
- We design a set of low-level southbound APIs that support atomic buffer operations and are composable for high-level APIs. The low-level APIs offer precision and efficiency, while high-level APIs allow applications to easily express where and how traffic flow should be paused and resumed.
- We build a proof-of-concept prototype of PB using open-source software, and develop three applications using PB. In our benchmarks, PB shows significant performance and scalability potentials, meeting or exceeding 5G Quality of Service (QoS) standards. In simulations, PB delivers near-optimal results and show huge improvement over control-plane buffering solution. For example, PB-enabled mobility

[1]Mobility at much higher frequencies, sometimes with longer handoff duration than interval, as expected in 5G networks. Will be described in detail in §2.5.1.

management delivers near-optimal (within 5% of the theoretical maximum throughput) results which more than double that of control-plane buffering; the decoupling between control and buffering allows PB to consistently outperform control-plane buffering in the connectionless communication use case by several orders of magnitude regardless of traffic volume.

## 2.2 Why Programmable Buffer?

We first draw a clear distinction between Programmable Buffers (PBs) and legacy switch buffers (queues). **Switch buffers (queues)** are part of the switch processing pipeline. They *temporarily* hold packets while packet schedulers decide in what order and when to serve these packets. They absorb arrival rate fluctuations to prevent packet loss, or enforce QoS metrics. **PB** is a storing unit alongside the switch processing pipeline. PBs serve as on-switch traffic sinks (in the case of "pausing") or sources (in the case of "resuming") when attached to the pipeline and they can hold the packets indefinitely unless otherwise instructed by the control-plane.

In many scenarios, packets are required to be buffered until a certain event happens (*e.g.,* mobile device reconnects to a new base station). Such events are typically not switch-local and require a global view to manage multiple buffers in different parts of the network in a coordinated manner.

Moving toward 5G and embracing different types of mobile devices and applications, we will see an increasing demand for buffering support. As SDN and NFV become the building blocks and key enablers for next-generation networks, we can leverage them to make network buffering programmable and provided as a general function for all applications just like forwarding and packet processing. This also echos with the vision of network slicing [44] which partitions network architectures into virtual elements with different requirements. Here we list three motivating examples to show that the requirement for network buffering varies depending on the application and device types. Therefore, catering

for application-specific use cases will not fit all and will end up reinventing wheels.

### 2.2.1 Network Buffering Examples

**Mobility Management.** In cellular networks, handling user mobility is a core service. Since packet loss and out-of-order delivery have a severe performance impact on TCP connections, the network needs to buffer (pause) users' downlink traffic before they disconnect from the base station, and resume it after they reconnect to another base station. End host applications are agnostic of such mobility or buffering behaviors, which are managed by the network control plane.

**Network Function Flow Migration.** With NFV, scaling a service simply requires the network to instantiate new NF instances. However, as pointed out in [42], many stateful network functions (*e.g.,* Bro IDS) require current states of the old NF instance to be transferred to the new instance, which must be completed prior to flow migration for correctness. This requires the network to actively buffer the live traffic while state transfer is happening.

**Connectionless Communication.** As 5G approaches, new communication paradigms such as device-to-device communications need to be supported efficiently. Such communications should work even when one end is offline or in mobility. Connectionless communication accommodates such special requirements by allowing the network to serve as a surrogate receiver, freeing the sender from buffering the data. This is even more crucial when the sender is battery constrained (*e.g.,* a sensor) and the receiver (*e.g.,* data analysis server) only goes online periodically. In this case, buffering requests originate from end hosts instead of the network.

## 2.3 PB Framework

### 2.3.1 Overview

PB framework extends from the typical SDN-NFV paradigm with a Buffer Engine on each data-plane node in coordination with a Buffer Service on the control-plane (See Figure 2.1). In addition to the default southbound APIs (interface 4 in Figure 2.1) such as OpenFlow [1] and P4 [31], Buffer Service communicates with Buffer Engine via PB's southbound APIs (interface 3). These low-level APIs are wrapped up by Buffer Service as higher level APIs for upper layer control applications (interface 1). Together with the built-in flow management capabilities (interface 2), a control application can create buffers when and where desired and direct traffic flows into/out of buffers.

Inline with the 5G vision, elements of the PB framework can be mapped to different functions in the 5G architecture: PB as User Plane Function (UPF), and Buffer Service as Session Management Function [45].

Next, we will break down the PB framework and describe the design of each in details. We start from the data-plane switch abstraction, Programmable Buffer Switch (PBS, Sec. 2.3.2). On the switch are the two programming abstractions, programmable buffer and virtual port. Then we introduce the southbound buffer APIs (Sec. 2.3.3) that are used to orchestrate and monitor the states of programmable buffer and virtual port. Buffers in different states (Sec. 2.3.4) carry out different functionalities. Control applications manage these states and fulfill their intentions through the northbound APIs (Sec. 2.3.5).

### 2.3.2 Programmable Buffer Switch (PBS)

In its essence, Programmable Buffer Switch is a buffer-enabled SDN switch. Besides the typical SDN switch composition (a pipeline of match+action tables, an on-switch agent/daemon, and external ports), PBS is comprised of programmable buffers (buffers) and virtual ports (vports). To manage the buffers and vports, a PBS implementation can choose

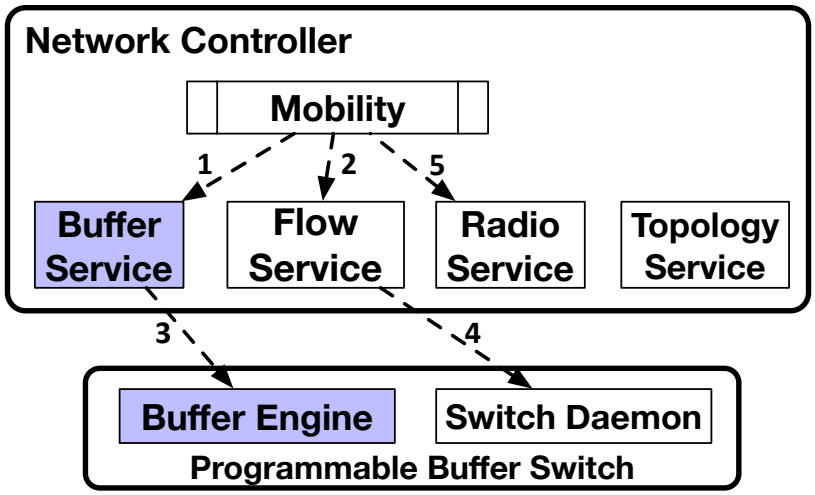



Figure 2.1: PB framework architecture

to either have a PBS agent (buffer engine) alongside the default switch agent (See Figure 2.2) or extend the switch agent with buffer/vport managing capabilities. As per the SDN paradigm, buffers, vports, and match-action table entries are dynamically created, configured and removed by control applications running on top of the network controller. Naturally, PBS can fall back to a regular SDN switch and be completely backward compatible with SDN applications that do not utilize PB APIs.

**Programmable Buffers**: Programmable buffers serve as data-plane storage buckets on PBS nodes. Distinct buffers have memory isolation and each buffer has an initial memory size specified at its time of creation and re-configurable later by control applications. To make them really instrumental, the controller must create vports that bind buffers to the switch (See buffer-1 and vport-1 in Figure 2.2). By default, buffers store packets in the order they are received and simply implement a FIFO queue. They can also be configured with other queuing polices (*e.g.,* priority queue). Buffers do not perform any manipulation of packet contents and any such manipulation is done by the switch processing pipeline (OpenFlow or P4) before the packets enter a buffer or after they exit a buffer.

**Virtual Ports**: Vports are software interfaces in PBS that bind buffers to a PBS node. Vports share the same 'port' abstraction with external ports, i.e., its one end is attached

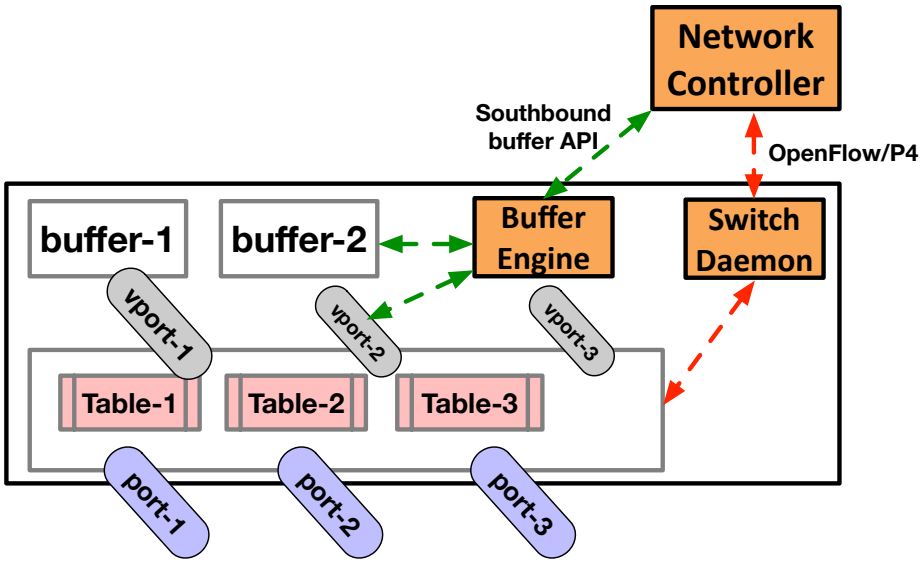



Figure 2.2: PBS architecture

to the switch processing pipeline and the other end is either attached to another entity (in this case a buffer) or free. Although a vport is bidirectional, we intentionally give each vport two modes of operations: *RX, TX*[2]. In *RX* mode, packets coming from the switching pipeline (following a flow table entry) will enter the buffer that is bound by the vport. In *TX* mode, packets in the buffer are sent to the switching pipeline. One could think of buffers as virtual hosts from the switch's point of view. Thus, it is possible to bind multiple vports to a buffer. Typically, however, one (in RX mode) or two (one in RX and the other in TX mode) vports are attached to a given buffer.

Vports and buffers themselves do not have the notion of what a "flow" is. Vports fill and empty buffers, while buffers queue packets without interpreting their headers or payloads. Therefore, by specifying a match-action table entry, the controller determines which set of packets should be sent to or retrieved from particular buffers. In each table entry, vports are specified as in-ports or out-ports depending on the flow direction.

Note that once a buffer is created and bound to switch by vports, the controller can

[2]Explicit mode configuration is actually quite useful: it allows the vports to have certain access control for traffic going through buffers. For example, when a vport is in *TX* mode, any packets coming from the pipeline to this vport will be dropped.

reuse the same buffer for any other network flow by simply modifying the match-action table entries. For instance, a buffer initially used to store all packets destined for mobile subscriber Alice can later be used to store all packets destined for mobile subscriber Bob. If desired, the same buffer can be used to store both Alice's and Bob's network traffic at the same time.

### 2.3.3 Southbound Buffer API

For southbound communications between the Buffer Service and Buffer Engines, we intend to make the APIs stable and atomic (composable), since they support the most fine-grained buffer operations.

As shown in Figure 2.2, Buffer Engine exposes programmability and monitoring capabilities to authorized external controllers through the PB southbound APIs. Table 2.1 shows the APIs to orchestrate the state of a buffer and vport, and to query/subscribe buffer-/vport status. The function of each API call is pretty self-explanatory. One important issue to point out is that the number of control messages transmitted through the southbound PB channel does not necessarily equal that of API calls, which could be quite large due to their fine granularity. By bundling several API calls in one control message, the overhead (delay) of actually performing that many API calls could be further reduced. For example, if a control application instructs the Buffer Service to create three buffers *B1,B2,B3* at switch *S1*, it generates only one control message instead of three. We assume no bundling throughout the rest of the paper and leave it for further study.

Next, we show the five typical states of a buffer and how each transitions into another. These state transitions are managed by the southbound buffer APIs.

### 2.3.4 Programmable Buffer States

There are five typical states of a buffer. Figure 2.3 depicts the simplest scenarios of each state.

Table 2.1: Southbound buffer APIs

| API | Parameters | Notes |
|---|---|---|
| create_buffer() | device_id, size, queue_type | queue_type: default to FIFO |
| create_vport() | device_id, mode, [port_num] | vport mode: {RX, TX} |
| bind_buffer_to_vport() | device_id, buffer_id, vport_id | Bind the given buffer to given vport |
| unbind_buffer_from_vport() | device_id, buffer_id, vport_id | Unbind the given buffer from the given vport |
| set_vport_mode() | device_id, vport_id, mode | Change the mode of the given vport |
| remove_buffer() | device_id, buffer_id | Remove the given buffer |
| remove_vport() | device_id, vport_id | Remove the given vport |
| query_buffer() | device_id, object, [rule] | object: what to query, e.g. buffer utilization. |
| query_vport() | device_id, object, [rule] | rule: notify subscribed application when met |

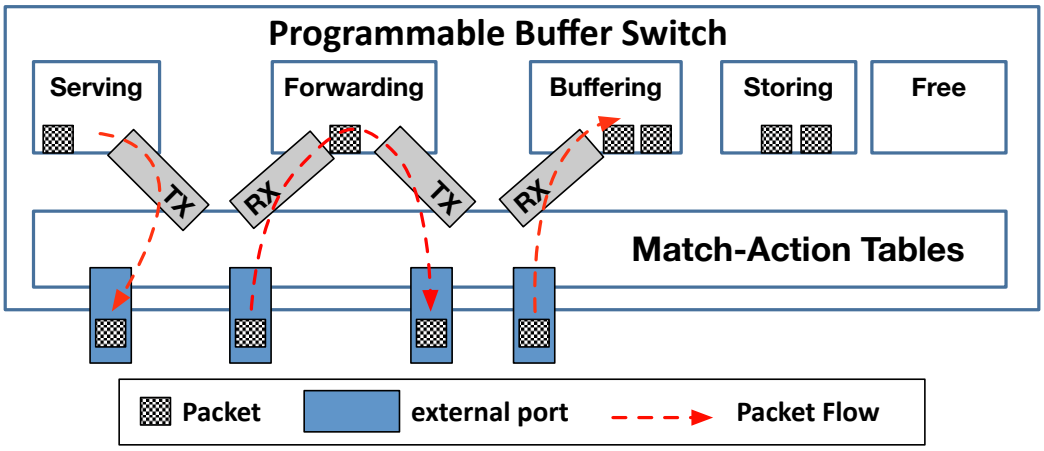



Figure 2.3: Five different states of a buffer

**Buffering State**: A buffer is bound to a switch by *only* RX mode vports, which means there is only inbound traffic towards the buffer. As mentioned in the last section, in order for traffic to be steered into the buffer, there should be match-action table entries specifying these vports as out-port. Inside the buffer, packets will be placed at the end of the FIFO queue unless otherwise configured by the controller. If the buffer exceeds its capacity, either new packets will be dropped (from the tail) or oldest packets will be dropped (from the head) based on the buffer configuration set by its control application.

**Serving State**: Opposite to the Buffering state, *only* TX mode vports are binding the buffer to the switch. Packets will be removed from the head of the FIFO queue, sent out via the attached vports, and processed through matching table entries.

**Forwarding State**: When *both* RX mode and TX mode vports are bound to the buffer, it is in the Forwarding state. Incoming packets will be placed at the end of the FIFO queue

while the oldest packets will be removed from the head.

**Free State**: If an *empty* buffer has *no* vports bound, it is in Free state. A buffer starts in Free state when first created; a buffer in Serving state transitions to Free state if it is completely emptied and unbound from vports. A Free state buffer is essentially a resource that can be recycled.

**Storing State**: If a *non-empty* buffer has *no* vports bound, it is in Storing state instead. A buffer in Buffering state transitions to Storing state if it is unbound from its vports. In this state, buffers hold the packets as long as the switch exists and the buffer is not removed by the control application.

Transitions between different states occur as a result of one or more API calls listed in Table 2.1. A buffer starts in the Free state with *create_buffer* command. It can transition to Buffering state or Serving state with *create_vport* (if needed) and *bind_buffer_to_vport*. A Buffering state buffer can transition to Forwarding state if a TX mode vport is bound, or to Storing state if all vports are unbound with *unbind_buffer_from_vport*, etc.

Note that, because of the explicit vport mode design, a **Forwarding State** buffer can transition to **Buffering State** by simply changing the *TX* vport to *RX* mode, and vice versa, without even modifying existing match-action table entries. As later shown in the buffer-enabled applications, this allows the controller to be minimally involved (sending one control message) and more scalable.

### 2.3.5 Northbound Buffer API

Common SDN controller implementations [29, 30, 46, 47] come with basic network services like topology discovery and flow management. These services provide northbound APIs to upper layer control applications for managing data-plane nodes while saving them from the troubles like discovering the topology or crafting a control message from scratch. Following the same paradigm, Buffer Service provides high-level buffer APIs to control applications to decide where, when and how a network flow is buffered.

These APIs can be divided into two levels: task-level and intent-level. **Task-level APIs** are directly composed by southbound buffer APIs that execute in a certain order to carry out a common task. In Figure 2.4, we present two task-level APIs that carry out two most common tasks in buffer-enabled applications: *create a buffer in a given state* and *change a buffer to a given state*. They are both solely composed by southbound buffer APIs introduced in §2.3.3. Buffer Service maintains a copy of the states of buffers and vports, thus in the second function *bufferStateChange* an application does not need to provide the current state of the buffer that it's operating on.

On top of the task-level APIs, we present two **intent-level APIs**: *pause_flow()* and *resume_flow()*. Both APIs composed of not only task-level buffer APIs, but also flow APIs.

*pause_flow(sw_id, in_port, flow_filter, [buffer_id])*. Through this API call, control applications decide where (i.e. which Programmable Buffer Switch) to buffer what flow coming from which port. Buffer_id is optional. If it is not provided, Buffer Service will automatically allocate an unoccupied buffer or create a new buffer. Depending on the status of the target flow, Buffer Service might add a new match-table entry to redirect the flow into the buffer, or, if the flow is going through a buffer already, simply change the state of that buffer to Buffering. And as shown in §2.3.4, changing a buffer from Forwarding state to Buffering state could be as simple as just setting the TX vport to RX mode.

*resume_flow(sw_id, out_port, flow_filter, [buffer_id])*. This API call allows control applications to turn Storing state buffers into traffic sources or resume flows in a Buffering state buffer by changing it to Forwarding mode. If a buffer_id is not provided, Buffer Service will try to locate the buffer used for storing the flow and redirect its content to the out_port. Otherwise, it will add one or more table entries for packets coming out of the given buffer. Note that flow classification happens twice in the second case, since flow filters used for *pause_flow()* do not necessarily need to match those used for *resume_flow()*. Applications could simply allocate a huge buffer to store flows coming from different sources, and later decide which sub-flows go to which destination.

```
// Task-level Northbound Buffer APIs
import pbService as pb

def createBuffer(swID, state, [buffer params...]):
  // create a new buffer
  buf = pb.create_buffer(swID, [buffer params...])
  // create vports and bind buffer to them
  switch state:
    case Buffering:
        vp = pb.create_vport(swID, RX)
        pb.bind_buffer_to_vport(swID, buf.id, vp.id)
    case Forwarding:
        vp1 = pb.create_vport(swID, RX)
        vp2 = pb.create_vport(swID, TX)
        pb.bind_buffer_to_vport(swID, buf.id, vp1.id)
        pb.bind_buffer_to_vport(swID, buf.id, vp2.id)
    // other cases...

def bufferStateChange(swID, bufID, state):
  buf = pb.getBuffer(bufID)
  switch buf.state and state:
    case Free and Buffering:
        vp = pb.create_vport(swID, RX)
        pb.bind_buffer_to_vport(swID, bufID, vp.id)
    case Buffering and Forwarding:
        vp = pb.create_vport(swID, TX)
        pb.bind_buffer_to_vport(swID, bufID, vp.id)
    case Buffering and Serving:
        pb.set_vport_mode(buf.vports[0], TX)
    case Forwarding and Buffering:
        pb.set_vport_mode(buf.vports[1], RX)
    case Buffering and Storing:
        pb.unbind_vport_from_buffer(swID, bufID, buf.vports[0])
    // other cases...
```

Figure 2.4: Task-level buffer APIs

## 2.4 Supporting Existing Apps

Network buffering is a critical function required by many existing applications. This section introduces two of these applications in more details and show how they can be supported in a PB-enabled network.

### 2.4.1 LTE Mobility Management (LMM)

As described in §2.2.1, LTE mobility management is complex and requires flow buffering and routing across multiple nodes. The initial phase involves radio signal measurements and reporting by the UE to its current base station (Source eNB). Source eNB makes handoff decision based on these measurements and requests handoff from a Target eNB. If request is admitted, the Source eNB starts buffering the downlink packets and instructs UE to establish a radio connection with the Target eNB. The Source eNB sends its buffered and in-transit packets coming from the Serving Gateway (S-GW) to the Target eNB. The Target eNB buffers these packets until radio connection is set up for the UE. In parallel, the Target eNB through the Mobility Management Entity (MME) performs a path switch from the S-GW to itself for all future downlink traffic. To preserve packet order, Target eNB buffers all the packets from the new path until all the buffered and in-transit packets from the source eNB are served. To facilitate the detection of last in-transit packet, S-GW transmits a special packet with End Marker. Once the marked packet is received by the Target eNB, it starts serving the buffered and in-transit packets from the new path. This whole process is in place to ensure loss-free, order-preserving packet delivery for good TCP performance.

In PB-enabled networks, the PBS data-plane abstraction applies to eNBs as well. LTE Mobility management (LMM) application runs on the controller and channel measurements from the UEs as well as load information from the eNBs are passed onto this application. LMM makes handoff decision, target eNB determination, and admission control based on these information. LMM sets up one programmable buffer at Source eNB, where the UE is currently attached to, and two buffers at the Target eNB, one for packets coming from Source eNB and the other for downlink traffic coming from the new path (Figure 2.6). All buffers are initially created in the *Buffering* state. Once buffers are set up, LMM instructs the UE to detach from the current eNB and attach to the new eNB. It instructs the anchor switch to switch from old path to new path while instructing Source eNB to change the state

```
// LTE Mobility Management with PB
import pbService as pb
import flowService as flow
import radioService as radio

self.on(HandoffStart, function(event) {
    // buffer 0 at source eNB
    buf0 = event.sourceBS.buffers[0]
    // buffer 1,2 at target eNB
    buf1 = event.targetBS.buffers[1]
    buf2 = event.targetBS.buffers[2]
    // direct traffic from old path to target eNB
    flow.FlowMod(buf0.vports[1], buf1.vports[0])
    // detach UE
    radio.detach(event.ue, event.sourceBS)
    // switch path at anchor switch
    flow.FlowMod(anchorSW.port[2], buf2.vports[0])
})

self.on(HandoffEnd, function(event) {
    // ue attaches to target eNB
    radio.attach(event.ue, event.targetBS)
    // turn buffer 1 into Forwarding State
    pb.set_vport_mode(buf1.vports[1], TX)
})

self.on(IndicatorReceived, function(event) {
    // turn buffer 2 into Forwarding State
    pb.set_vport_mode(buf2.vports[1], TX)
})
```

Figure 2.5: PB-enabled LMM application

of its buffer to *Forwarding*. At this point, all the buffered and in-flight packets coming to Source eNB are diverted toward the first buffer at Target eNB (green arrow in Figure 2.6). The second buffer at Target eNB is still in the *Buffering* state and hence all new path packets are buffered in this second buffer. Once the radio link is established between UE and Target eNB, LMM receives the completion signal upon which it instructs Target eNB to change the first buffer state to *Forwarding*. Hence, two buffers at Source and Target eNBs are in tandem serving UE the in-transit traffic coming via old path. To ensure old path is cleared, when switching the path, LMM also injects an indicator packet at anchor switch

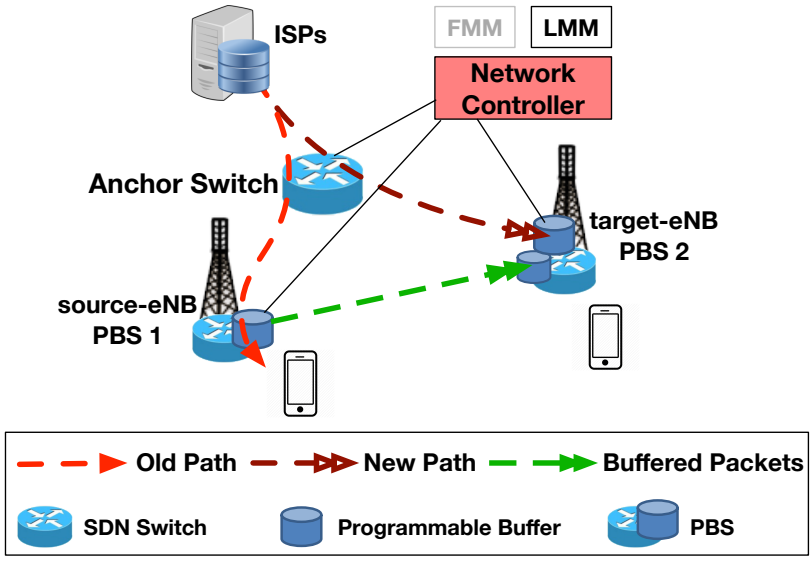



Figure 2.6: PB-enabled LTE Mobility Management

(for example, using a packet-out message in OpenFlow protocol). The indicator packet traverses anchor switch and Source eNB before reaching Target eNB as the last in-transit packet from path A. Since this packet is marked and every eNB is programmed to notify the controller upon receiving it, LMM knows that there are no in-transit packets left from path A. Now, LMM instructs Target eNB to change the state of the second buffer to *Forwarding* state. UE starts receiving packets from path B. Figure 2.5 shows part of LMM in pseudo code.

### 2.4.2 NFV Flow Migration

NFV flow migration can be considered as a special version of mobility management, as the mobility happens inside the network instead of network edge. This use case is specifically addressed in [42]. The authors first started using the SDN controller to buffer in-flight packets during NF state transfer, which leads to triangular routing and many other performance and scalability issues. They later adopted an alternative approach [48], which instead requires the NF instance to buffer in-flight packets before state transfer finishes.

With PB's support, the controller only needs to set up a Buffering State buffer for

each new NF instance at the same switch, and chain the buffer with its corresponding NF instance with flow rules. This way, the buffer can be independently managed by the controller and adjusted according to different traffic volume and pattern without having to modify the NF programs. Packets will stay inside the buffer and be immediately available when state transfer finishes.

## 2.5 Enabling New Apps

With PB's APIs and abstractions, we can achieve more than supporting existing applications. In this section, we introduce two new network applications that PB enables: Fast Mobility Management and Connectionless Communications.

### 2.5.1 Fast Mobility Management (FMM)

With 5G envisioning massive bandwidth improvement over 4G, the current radio access link technology in LTE networks is no longer viable. This has researchers look into alternatives such as Millimeter-Wave (mm-wave) and much denser cellular deployments [49]. Abundant spectrum of high frequencies and densification resolve the bandwidth shortage problem, but such systems also require high directionality and narrow beam widths, imposing major challenges in mobile scenarios. Consider the scenarios where many roadside or lamp-post base-stations are deployed with 10 to 20 degree beam-widths. The beam-training takes 10s of milliseconds [50], and even at moderate vehicular speeds (e.g., 30 mph) with 5-meter separation between the base station and road lanes, the residence time in each base station becomes comparable to the beam-training time.

This becomes problematic for the LMM application. In that, after each handoff, the second buffer at the target eNB (See Figure 2.6) has to wait till the first buffer is emptied (old path is clear) before it can start serving the UE. In theory, the time it takes for the first buffer to empty is equal to or larger than the handoff duration, because the first buffer stores all in-flight packets during the handoff period. In practice, due to control latencies,

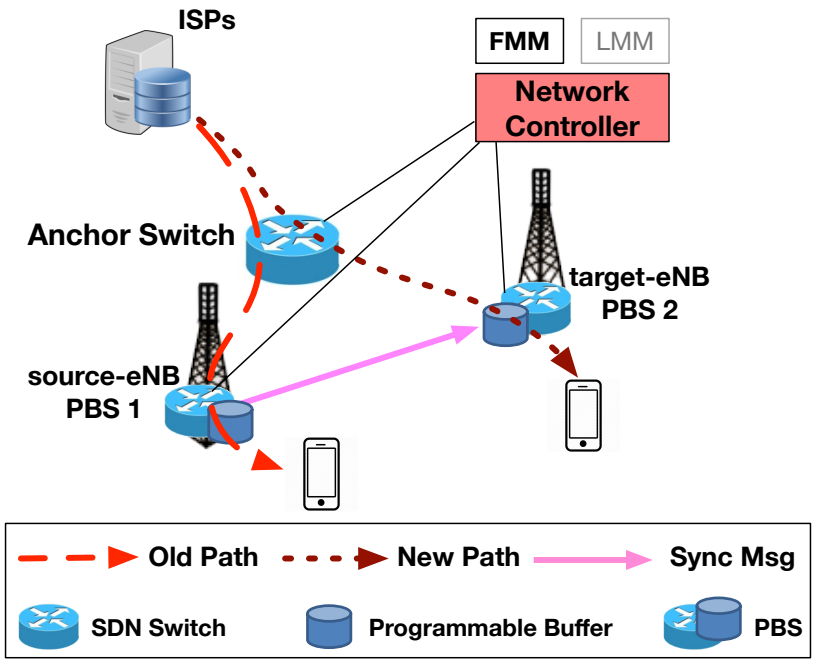



Figure 2.7: PB-enabled Fast Mobility Management

this time is even longer. Simply put, LMM will not work under such extreme conditions because residence time with a eNB could be shorter than handoff duration. Therefore, we propose a new mobility management solution that eliminates inter-eNB traffic forwarding and supports much higher handoff frequencies, called Fast Mobility Management (FMM).

This application takes a more aggressive approach to ensure packets are always ready whenever and wherever a UE attaches, and it can detach anytime it wants. To enable this, when a UE is attached to a particular eNB, all neighboring eNBs (or a subset of them based on predictions of mobility pattern) will be receiving downlink packets from the anchor switch and buffering them. In other words, the anchor switch is multicasting UE packets to all potential target eNBs. When the UE moves and reattaches to another eNB, the buffer there can immediately start serving the UE without having to wait for the source eNB to forward the buffered packets. Without buffer programmability, this kind of dynamic service cannot be orchestrated unless the software in each base station is upgraded. With PB, however, this application can be easily supported as follows.

The FMM application provisions a buffer in *Forwarding* state at the UE's current attachment point (i.e., Source eNB-PBS). At the same time, FMM provisions buffer for the same UE in *Buffering* state for each potential next base station (i.e., Target eNB-PBS nodes). FMM also installs a forwarding rule at the anchor PBS node of all these base stations to multicast the UE traffic to Source and Target eNB-PBS nodes before any handoff decision is taken. Thus, packets are buffered at Target eNB-PBS nodes. Before UE starts detachment, FMM changes the state of UE's buffer at Source eNB-PBS to *Buffering*. After the reattachment, FMM changes the state of UE's buffer at the new eNB-PBS to *Forwarding* and UE can start receiving packets from it. After the handoff, FMM can update the set of Target eNB-PBS nodes, thus accordingly change the multicast group at anchor PBS while terminating/recycling the buffers provisioned for UE at eNB-PBS nodes that are no longer potential targets.

**Inter-buffer synchronization**: Since in FMM each buffer keeps their own copy of the packets, packet-loss or duplicates become an issue. For example, if the target-eNB buffer is sufficiently large, we should expect the first packet in it to be older than the head-of-line packet of the TCP session. In other words, there will be duplicate packets in the target buffer. In contrast, if the target buffer is too small, there will be packet loss. Both duplicates and losses can lead to inferior TCP performance. To resolve this problem, there needs to be a synchronization mechanism to align the target buffer head with the source buffer head. That is, when handoff happens, as the source buffer stops sending, the target buffer should know what the last sent packet was and purge any packets older than it. Obviously, this only works when the target buffer is sufficiently large (has duplicates), since there is no way to recover lost packets. Such synchronization only needs to convey a unique packet identifier from the source buffer to the target buffer, which is negligible compared to LMM's traffic redirection. In our experiments, we find 2 bytes of IPv4 id and 2 bytes of transport layer checksum to be reliable for uniquely identifying packets even when encrypted. Since control applications can decide which buffer implementation to use

for their traffic, they can choose the right identifiers based on the traffic pattern. Figure 2.7 depicts a handoff scenario with FMM and synchronization enabled.

**FMM vs. LMM**: In the 5G era, as network slicing [44] becomes the norm, different mobility management solutions like FMM and LMM are expected to run on the same infrastructure serving different devices and users based on their needs. Compared with LMM, FMM allocates more buffers for each user since it uses multicast at the anchor switch to ensure immdediate packet availability, which has higher buffering overhead. Therefore, FMM targets a small portion of users that are travelling at a high speed and thus handoff much more frequently. In terms of control overhead, FMM generates the same amount of control messages during each handoff as LMM (managing the multicast group is outside the control loop of handling handoffs).

### 2.5.2 Connectionless Communications

Connectionless network services which are, e.g., used to support Internet of Things (IoT), have been one of the key use cases discussed for next generation mobile networks [51]. PB abstractions can be utilized by connectionless services to have a slice of the underlying transport fabric as a caching and content distribution infrastructure and enable asynchronous communications between devices (device-to-device communications).

To store/upload any content, the control application first associates the content with a network flow. How this association is done using which packet header fields is implementation specific. Once the one-to-one association between contents and network flows is done, to store a particular content on a given PBS node, control application should create a distinct buffer for the content (i.e., for the corresponding network flow) at the given PBS node in *Buffering* state. After it ensures that all the flow packets are stored, the control application can transition the buffer into *Storing* state (Figure 2.3).

If stored content is requested by another node in the network, the IoT or content distribution application first sets up a routing path and simply transitions the buffer of that

content into *Serving* state, which will automatically start serving the requesting node.

## 2.6 Evaluation

In this section, we present our prototype of PB using open-source software and its scalability and performance gains compared with alternative SDN solutions.

### 2.6.1 Prototype and Methodology

We prototype **Buffer Service** as a Ryu [29] controller module which provides the northbound buffer APIs to other applications. We implement **Buffer Engine** in C++ as a process running alongside the software switch. Buffer Service establishes connection with Buffer Engine via gRPC [52]. The engine serves as both gRPC client and Docker agent. **Programmable buffers** are packaged as Docker [37] containers managed by Buffer Engine via Docker APIs. Each container runs a C program that receives IPC calls (*e.g.,* bind/unbind) from Buffer Engine. The motivations behind containerizing buffers, besides ease of managing resources like CPU and memory, is that control applications can choose between different buffer implementations by specifying a Docker image. This allows different queue types and application-specific tweaks (*e.g.,* inter-buffer synchronization in §2.5.1) to be pre-built and used flexibly.

We use two open-source **Software Switch** packages, OpenvSwitch [38] and mSwitch [53], and two companion **Virtual Port** implementations, TAP [54] interfaces and mSwitch ports respectively. OpenvSwitch is OpenFlow-compatible and thus used in our simulations (§2.6.3); mSwitch is picked for its high port density and used in scalability tests (§2.6.2.2). We also implement two packet processing techniques for programmable buffers: Memcpy and Zerocopy. Memcpy, as its name suggests, uses the Memcpy() function to copy packets from/to vport packet queues. Data copy incurs high overhead under high bit rates. We thus implement a zero-copy programmable buffer with netmap [55] that allows us to preserve packet buffers out of packet I/O queue without data copy. We use this

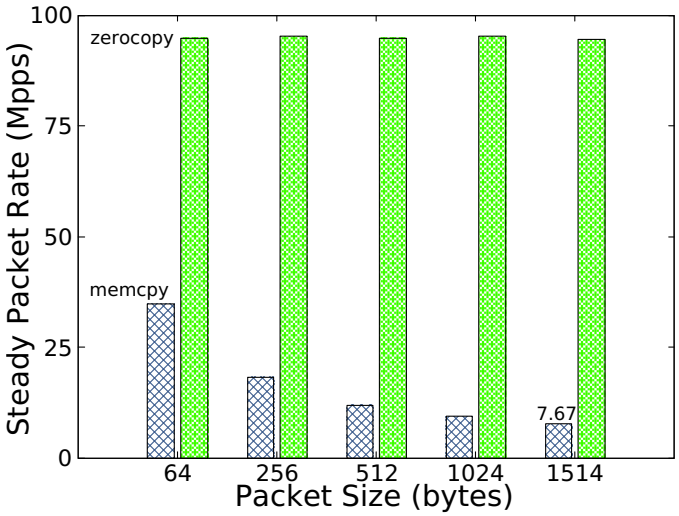



Figure 2.8: Buffer packet rate with varying packet sizes

variant of programmable buffer on top of mSwitch, and demonstrate superior performance in §2.6.2.2. We also modify mSwitch to not perform data copy between its ports to further reduce the overhead. Programmable buffer size is set to 1024 packets for benchmarks and 10000 for simulations. Unless otherwise specified, we use an Ubuntu 16.04 (Linux 4.13.0) Desktop with Intel core i7-7700K@4.2GHz quad-core processor and 16GB of RAM for our experiments.

### 2.6.2 Benchmark Results

It is widely acknowledged that future 5G networks should be able to support high-bandwidth applications like 4K video streaming, Virtual Reality and Augmented Reality while incurring ultra-low latency ([40, 56, 44]). Many believe the user experienced data rate should be at least 50Mbps and up to 1Gbps depending on coverage and resource availability. According to [57], for 5G radio access network, the control-plane latency should be less than 10ms and data-plane latency should be around 1ms. The decoupled design of PB allows it to scale up independently on the control-plane and data-plane depending on the workload. To see if each component of PB can meet these QoS metrics, we run several benchmarks as described below.

#### 2.6.2.1 Packet Rate and Latency

Since programmable buffers are on the critical path of user traffic, we perform the following experiments to get the steady packet rate and one-way latency of single programmable buffer with varying packet sizes. On the machine, we create a *Forwarding* state buffer, two pkt-gen applications (from netmap) as traffic source and sink, and connect them with two virtual ports. We set the two pkt-gens to transmit and receive mode respectively, and specify different packet sizes in each run. The three processes are pinned to three CPU cores, and the batch size is set to 512. We configure the virtual ports to operate as netmap pipes[3] and there are no packet losses. The results are shown in Figure 2.8. As expected, buffer with zerocopy has constant high packet rate (90+ Mpps), while memcpy incurs higher overhead as packet size increases, though still achieving 90+ Gbps throughput with 1514-byte packets. This indicates that both programmable buffer implementations are fairly efficient.

Similarly, we measure the one-way packet delay of programmable buffers by setting the pkt-gens ping and pong mode respectively (packets travel a round trip and the RTT is calculated at the ping side). We also use various packets sizes up to 1514 bytes. The results are shown in Table 2.2. For both zerocopy and memcpy versions with packet size up to 1514 bytes, programmable buffers incur no more than 6 microseconds one-way packet delay, which is more than two orders of magnitude smaller than the 1-ms 5G standard [57].

#### 2.6.2.2 Data-Plane Scalability

As 5G aims to serve massive number of devices, scalability becomes one of the most critical criteria when evaluating the design of PB. We start our scalability analysis by measuring the resource footprint of programmable buffers which correlates with the number of users one PBS can serve. In this test, we use the same setup as in the last benchmark (with buffer on path, 1514-byte packets), but instead of saturating the bandwidth, we vary the

[3] A shared memory packet transport channel supported by netmap.

Table 2.2: PB latency measurements

| | | |
|---|---|---|
| One-way Delay ($\mu$s) | Memcpy | 5.28 - 6.00 |
| | Zerocopy | 5.43 - 5.82 |
| API Call Execution Time (ms) | create_buffer | 39.9 |
| | create_vport | 11.7 |
| | bind_buffer_to_vport | 0.4 |
| | set_vport_mode | 1.5 |
| | unbind_buffer_from_vport | 1.4 |
| | flow_mod | 1.3 |
| | remove_vport | 27.0 |
| | remove_buffer | 39.7 |

throughput and measure the corresponding CPU usage. Our results show that at 40Gbps throughput, the memcpy version of buffer consumes 48.2% CPU while zerocopy version consumes only 5%. Both versions consume 0% CPU when idle, which is expected.

To better understand how well PBS scales to larger number of users (flows), we carry out another experiment to simulate up to 2048 active users (flows) simultaneously. Due to inherent CPU core requirements to host a large number of programmable buffers, we employed another Linux (4.11) server with Intel Xeon E5-2690v4@2.6GHz 14-core processor and 64GB of RAM (4 cores assigned to traffic source, 4 to buffers and 1 to traffic sink). Each flow is handled by a different buffer, totaling up to 2048 programmable buffers on one PBS instance. We write a simple routing module for mSwitch that (1) distributes incoming packets (from traffic source) to each buffer based on destination IP address, and (2) aggregates packets to one port (traffic sink). We use mSwitch with this module enabled to connect traffic source and sink with up to 2048 buffers, and record the aggregate throughput as shown in Table 2.3. Although the absolute values could vary on different machines, these results clearly demonstrate the data-plane scalability of PB.

**Finding 1.** PBS can deliver 100+Gbps throughput with over 2000 flows using 4 cores on a commodity server, attaining 50 Mbps per flow that is twice the recommended bandwidth for 4K video streaming [58].

Another factor contributing to programmable buffer's resource footprint is memory.

Table 2.3: PBS scalability with concurrent Flow(Buffer)s

| # of Flows/Buffers | 4 | 16 | 64 | 256 | 1024 | 2048 |
|---|---|---|---|---|---|---|
| Aggregate Lossless Throughput (Gbps) | 187 | 123 | 121 | 110 | 105 | 102 |

PBs have marginal memory overhead on top of what is needed for storing the packets. As such the memory requirement becomes simply the product of forwarding speed and the duration of traffic interruption. E.g., for 100 Gbps bandwidth, it will be several to tens of GBs. We argue that the amount of RAM available on a commodity server (*e.g.,* 128GB) is more than enough to satisfy the memory requirements for traffic interruptions at the scale of tens [59] to hundreds of milliseconds [42]. Further, PB dynamically (re)allocates buffer memory, unlike existing solutions with pre-determined buffer size based on estimation [60].

#### 2.6.2.3 Control-Plane Scalability

In the second row of Table 2.2, we show the RTT measurements of each PB API call and FlowMod [61] of OpenFlow. To get the inherent latency numbers, we run both the controller and PBS on the same host to minimize communication latency. Our test application repeatedly calls each southbound API and measures the time to complete each call. In our measurements, operations related to create/remove buffers/vports have higher overhead than simply binding vports to buffers and modifying vport modes. This is expected since the former operations involve memory (de)allocation. As mentioned earlier, buffers and vports have minimal idle resource footprints, thus provisioning them beforehand can effectively remove their presence on the critical path of latency-sensitive operations. Compared to FlowMod, other buffer API calls have similar or smaller latencies. As discussed in §2.8, many previous works have focused on improving the scalability of SDN control plane to support carrier-grade workloads [47, 62, 63], which are complementary to PB's design. For existing SDN applications, PB is fully backward compatible and thus shares the same level of scalability; for buffer-based applications (such as LMM shown in Code Sample 2.5),

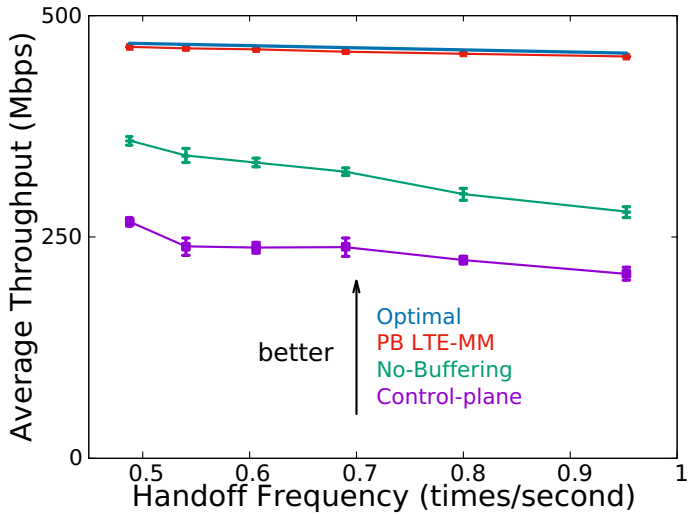



Figure 2.9: Average throughput of different LTE mobility management solutions on low handoff frequencies

since buffer API calls have similar costs, and the number of these calls is minimized[4], the control-plane scalability is not much impaired.

### 2.6.3 Application Performance

#### 2.6.3.1 LTE Mobility Management

As described in §2.4.1, PB allows critical core services to be virtualized while keeping low-latency data-plane buffering functionalities. One of the features PB offers is loss-free order-preserving packet delivery, which is fundamental for high TCP throughput and critical to many web services [64]. We implement the LTE mobility management application using *only* PB APIs and basic SDN flow management APIs (i.e. FlowMod in OpenFlow). For comparison, we also implement two alternative solutions that do not guarantee loss-free order-preserving delivery: one buffers packet at the control-plane, one does not buffer any packet. The control-plane buffering solution is what current SDN frameworks would adopt due to lack of buffering abstractions for the data-plane; the no-buffering solution serves as

[4]As explained in §2.3.4, PB's vport mode design allows applications to change buffer state by simply changing the mode of a vport.

a baseline with only TCP retransmissions.

We compare these solutions by simulating the simplest handoff scenario: one UE with an active TCP session detaches from one eNB and, after a set duration (50ms, typical LTE handoff duration [65]), reattaches to a new eNB where the TCP session is resumed (See Figure 2.6). The UE side (simulated wireless link) bandwidth constraint is 500Mbps[5], and the end-to-end latency is 20ms. Server runs an iPerf process. Consecutive handoffs are simulated with UE moving back and forth between two eNBs. Handoff intervals range from 1s to 2s, which corresponds to a handoff frequency from 0.5/s to 1/s. The results are shown in Figure 2.9. We calculate the optimal number by assuming no throughput recovery delay.

**Finding 2.** PB-enabled LTE Mobility Management yields near-optimal result for pedestrian and vehicular speeds in small cell environments.

#### 2.6.3.2 Fast Mobility Management

As described in §2.5.1, the Fast Mobility Management application supports high mobility by multicasting at the anchor switch. We implement the FMM application with PB APIs and SDN flow APIs similar to LMM. As mentioned in §2.6.3.2, we implement a simple and efficient synchronization mechanism inside FMM's buffers. When the source buffer's TX mode vport is changed to RX (*Forwarding* to *Buffering*), a special packet containing the identification information will be sent to the target buffer and be used to purge duplicate packets. In microbenchmark this mechanism incurs less than 1ms delay for comparing and purging 10,000 packets, which is one order of magnitude smaller than the handoff durations. We carry out high-frequency handoff tests with the FMM application and calculate the average throughput under each setting. We selected 10ms as the handoff duration, and

[5]Despite the term LTE mobility management, this scheme will be integrated in the 5G infrastructure as 5G embraces different access technologies to ensure seamless user experience. Therefore 500Mbps is not an overkill.

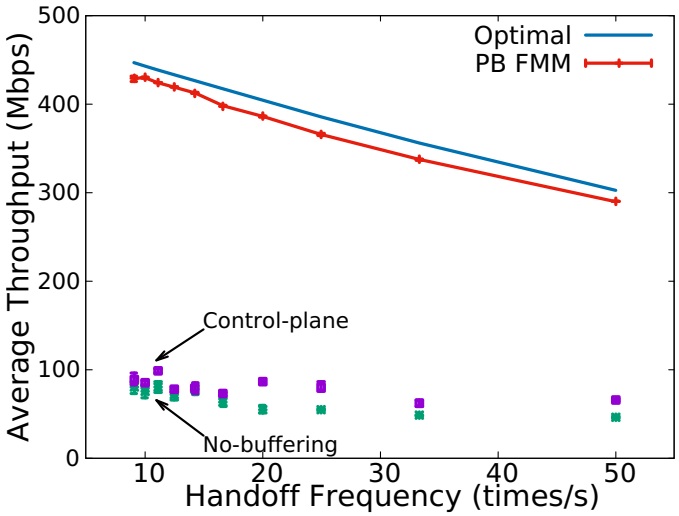



Figure 2.10: Average throughput of fast mobility management solutions on high handoff frequencies

ten handoff intervals ranging from 10ms to 100ms. In this way, we can see how FMM performs when handoff duration is larger than handoff interval. These settings combined give us handoff frequencies ranging from less than 10 times/s to 50 times/s. Results are shown in Figure 2.10.

**Finding 3.** PB-enabled Fast Mobility Management solution delivers near-optimal throughput with 5% or less overhead for ultra high handoff frequencies.

#### 2.6.3.3 Connectionless Communications

Both mobility management applications are data-plane intensive with mild control-plane workload. In contrast, connectionless communication (CC) application is less latency sensitive due to its asynchronous nature and less throughput hungry, yet imposes much higher control-plane overhead. An overwhelming number of IoT devices generate periodic burst of traffic and requires the control-plane to handle these events efficiently. We evaluate this scenario by running simulations on a CC application that controls 1000 data sources and one data sink. To simplify the setting, we set up one PBS instance con-

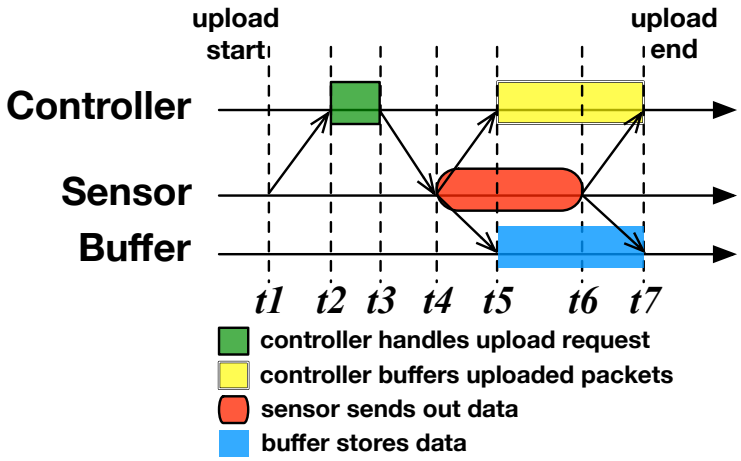



Figure 2.11: Upload process timeline

nected to 1000 virtual hosts working as ”sensors” and another virtual host that periodically collects the ”sensor” data. In each run, the ”sensors” send an upload request to the application, which assigns a small buffer for each of them and installs the proper flow rules that direct the uplink traffic into each buffer. After all data has been collected and stored in the network, the server node sends a download request to the application, which changes the state of each buffer and directs all traffic toward the server node. We record the timestamp for each stage throughout the process, such as ”upload request sent”, ”upload starts”, ”download finishes”, on both end hosts and controller. For comparison, we also implement a control-plane solution that stores all ”sensor” data on the controller. To see if data volume affects the result, we give each ”sensor” two files to send: one’s size is 10KB and the other 1MB.

Figure 2.11 shows a timeline of the upload process in both solutions. For simplicity, here we assume both upload processes take the same time to finish[6]: *t7-t1*. For the CC application, because control and buffering are decoupled, the controller only handles one upload request and is occupied for *t3-t2* which is constant. In contrast, the control-plane buffering solution adds another *t7-t5* which grows linearly with file sizes.

[6]In reality control-plane buffering takes much longer due to limited control plane bandwidth

In our simulations, each upload request takes around 0.53ms to process, while the control-plane buffering process takes 82ms for the 10KB file and 2 seconds for the 1MB file. The download request takes around 0.54ms to process, and the download process lasts 1ms and 85ms respectively.

**Finding 4.** PB-enabled Connectionless Communication application handles upload requests with constant overhead, outperforming control-plane buffering by 160 and 4000 times for 10KB and 1MB files respectively.

## 2.7 Discussion

**PB's applicability outside 5G and API generality.** Even though PB is strongly motivated by the use cases and visions of 5G, we believe that PB could potentially be useful in other scenarios as well, such as data center. The results shown in §2.6.2 also demonstrate this potential. The APIs are designed independently of the applications. In fact, two of the three use cases (Fast Mobility Management and Connectionless Communications) are developed after the APIs are finalized. We believe the APIs are fairly general as it captures the three critical elements in controlling flow buffering: What (flow matching), When (event triggered) and Where.

**Hardware switch compatibility.** PB's control-plane APIs and abstractions are independent of its data-plane implementations. That being said, the current implementation is only compatible with software switches because it relies on containers. More importantly, commodity servers have enough memory to satisfy the buffering requirements in high throughput settings (§2.6.2.2. A hardware switch-based implementation (*e.g.,* Tofino switches [66]) will likely also use commodity servers due to RAM limitations.

**Security considerations.** As user packets are kept indefinitely inside buffers, certain access controls need to be in place to protect user privacy. For example, applications should not have direct access to buffers&vports not created by them (unless they are recycled by

the controller), and buffers should be emptied before reused. Malicious devices could also try to launch Denial-of-Service (DoS) attacks by tricking applications into allocating exceptionally large buffers. We believe there are both opportunities and challenges with Programmable Buffer in the security area, and we leave them for future explorations.

## 2.8 Related Work

**Switch buffer management** is a well-studied topic dating back to the 70s [67]. Most previous works focused on designing packet scheduling algorithms or switch buffer architectures for specific use cases. More recently, to address the lack of programmability of packet scheduling, Sivaraman *et al.* [68] proposed a push-in first-out (PIFO) queue as a basic abstraction for programmable scheduler; Kogan *et al.* [69] proposed a framework for constructing custom switch buffer architecture; HotCocoa [70] implemented entire congestion control algorithms in programmable NICs. As explained above, programmable buffers and switch queue have different purposes. Switch queues are not designed to hold packets indefinitely and packet schedulers have no visibility beyond switch-local events. In addition, these solutions require new scheduler/architecture/algorithm to be recompiled which changes the switch processing pipeline.

**On-switch storage.** NetCache [71] proposed to use hardware programmable switches as key-value caches for load balancing storage server requests. In particular, the authors utilize the register arrays implemented with on-chip memory to store *small* values. Such hardware switches have limited/fixed number of registers and register size (128B) that are not well suited for PB which dynamically manages the number and sizes of buffers to store packets. In this work, reading and writing the cache are triggered by certain packets arriving at the switch, whereas PB allows the controller to programmatically turn a buffer into traffic source or sink.

**Network storage.** Plank *et al.* [72] presented the Internet Backplane Protocol for network applications to actively utilize storage as part of the network fabric. This work

differs from PB in many ways. Firstly, it is a distributed approach while PB exercises SDN's centralized model. Secondly, it treats network storage as a passive function triggered only by end-hosts, which cannot address scenarios where buffering necessity originates from within network (e.g. user mobility). Thirdly, end hosts lack the global view to optimally decide where to store. However, this paper does share the same vision with PB of the importance of in-network buffering for many applications.

**Packet buffering in SDN.** Without data-plane buffering support, SDN applications have to rely on the controller to temporarily store the packets [42], thus placing it on the critical path of network flows. This could incur high per packet latency [48], and occupy limited controller resources (memory, CPU and control channel bandwidth) [73], which suffers on scalability. PB addresses this problem by decoupling the control from buffering and keeping latency-sensitive operations on the data-plane: the controller is not on the critical path of network flows and is only involved to initiate "pause" or "resume" instructions (which, as shown in §2.3.4, are often accomplished with only one control message).

**Cellular SDN/NFV network architectures.** Recently, there has a been substantial focus from academia and industry on realizing the network architecture of 5G cellular core networks based on the SDN/NFV concept [74, 75, 76, 77, 78]. In particular, SoftCell [63] departs from the centralized policy enforcement in the core network by directing users' traffic through distributed middleboxes. SoftMoW [62] builds a scalable control plane using the hierarchy technique to enable global network optimization. These SDN/NFV architectures are complementary to PB in terms of scaling the control-plane and handling failures.

## 2.9 Summary

We propose a new yet fully backward-compatible SDN solution to allow applications to manage available memory on software switches for orchestrating where, when and how network flows are buffered. We show that the proposed Programmable Buffer (PB) ab-

straction can effectively expand programmability of existing SDN systems to allow further decoupling between the SDN control-plane and data-plane. We demonstrate that PB can be leveraged to provide mobility management, as it is done in the current LTE networks, with near-optimal performance. PB significantly outperforms alternative SDN solution that uses control-plane for buffering purposes. Benchmarks show great performance and scalability potentials, for it exceeds 5G standards in every aspect tested. Programmable buffers can easily deliver 90+ Gbps throughput with large packets, and scale out on Programmable Buffer Switches with acceptable overhead. Moreover, the PB abstraction is powerful enough to support a new mobility management scheme that handles extreme mobility scenarios with $<5\%$ performance drop.

# CHAPTER III

# Egret: Simplifying Traffic Management for Physical and Virtual Network Functions

In the previous chapter, we study the in-network buffering behavior of network applications and expand SDN *programmability* to efficiently support it. In this chapter, we look at one of the most common network application behaviors, which is to migrate traffic to and from network devices. We show how limited *generality* in existing SDN systems, which for the most part use programmable switches to steer traffic, can lead to increased complexity in development as well as operation of workflows that include traffic migration in a heterogeneous network. To tackle this problem, we study 205 change management Methods of Procedure (MOPs) from a major U.S. carrier and show that generalizing traffic migration with a unified model is feasible. Based on our findings, we develop *Egret*, a generic traffic migration system based on the SDN paradigm that uses abstraction and modularization techniques to simplify traffic management from both execution and implementation perspectives. Leveraging a novel mask-based abstraction for traffic migration jobs, Egret can further simplify reverse traffic migration and enable job parallelization.

## 3.1 Introduction

Despite the advancement and growing adoption of SDN, it remains challenging to perform traffic migration on a heterogeneous data plane, because different types of network functions have (or depend on) drastically different traffic steering capabilities/methods. For example, a router can easily drain its traffic by increasing OSPF weights of its links, but it is incapable of controlling what flows to drain or specifying destinations for the drained traffic; in contrast, a firewall often has to rely on external functions like load balancers or switches to steer traffic which do support more fine-grained controls.

These differences lead to highly customized solutions that tailor to specific applications (*e.g.,* disaster mitigation [79]) and/or network setups (*e.g.,* load-balancers and programmable switches). Be they traditional MOPs or systematic approaches [79, 42, 80], specialized solutions often require excessive knowledge of the infrastructure to develop and use (*e.g.,* topology, device type). As NFV becomes more prominent and network data plane more heterogeneous and dynamic, developing and using specialized solutions will inevitably become unsustainable and costly due to their poor reusability across applications and network functions.

We believe generalization is key to fundamentally addressing this problem because a unified and extensible model will greatly simplify (1) development of solutions by vendors (extension over customization) and (2) execution of solutions by operators (standardized interfaces over proprietary interfaces). To this end, we conduct a study on 205 change management MOPs (§3.2.1) from a major U.S. carrier to identify the commonalities and differences in traffic migration for different network functions in different scenarios. We find that:

(1) Four key parameters, *Target*, *Peer(s)*, *Weight(s)* and *Filter(s)* are sufficient to describe any traffic migration intentions. *Target*, around which traffic migration is performed, is the only required parameter (§3.2.3). (2) All traffic migration methods can be categorized as either local or remote. Local traffic migration takes place on the *Target* (*e.g.,* a

router steers traffic through OSPF weight adjustment), whereas remote traffic migration takes place on external network components such as load-balancers, DNS servers or programmable switches, which we refer to as *Anchor Points* (§3.2.4). (3) Major sources of complexities of traffic migration come from migration method & anchor point discovery, target & anchor point configuration, reverse traffic migration and parallel job coordination (§3.2.5).

Based on these findings, we design *Egret*, a generic traffic migration system that simplifies traffic management for different network functions. Egret decouples specification of traffic migration intentions from intrinsic configuration details with its generic interfaces. It modularizes common stages of traffic migration workflows to enable plug-and-play by vendors. To further reduce the complexity of managing states of traffic migrations, Egret uses a mask-based abstraction to efficiently keep track of individual jobs to simplify reverse traffic migration and job parallelisation.

We make the following contributions[1] in this chapter:

- We conduct the first comprehensive study of traffic migration as a general procedure through extensive analysis of 205 change management MOPs from a major U.S. carrier. We identify common patterns, stages and key parameters that help define a unified model of traffic migration. We also shed light on major sources of complexities in existing traffic migration practices.
- We propose the design of Egret, a traffic migration system that simplifies traffic management for different network functions for both operators and vendors through generalization, automation, and modularization.
- We prototype Egret and integrate it with the control-planes of routers, load-balancers and programmable switches. We demonstrate through high-fidelity emulations how Egret simplifies traffic migration on a heterogeneous data-plane.

[1]Part of this work was conducted in collaboration with AT&T, including §3.1, §3.2, §3.3, and §3.4.3. §3.4.1 and §3.4.2 are completed independently outside the collaboration.

## 3.2 MOP Analysis

To better understand how different types of network functions perform traffic migration in practice, we perform an extensive analysis of 205 change management Methods of Procedure (MOPs) from a major U.S. carrier.

In this section, we first give an overview of the MOPs (§3.2.1), then we select three representative examples of those MOPs (§3.2.2) to help illustrate our findings (§3.2.3,§3.2.4,§3.2.5).

### 3.2.1 Method of Procedure

Methods of Procedure (MOPs) are manuals/documents that provide step-by-step instructions on how to perform an operation. Our study focuses on change management MOPs that describe the process of imposing changes such as software upgrades to specific network components. This process usually contains locking, health checks[2], state/configuration preservation, traffic migration, change deployment, reverse traffic migration, state/-configuration restoration, and unlocking. In this work, we focus on the traffic migration procedure of these MOPs.

### 3.2.2 Representative MOP Examples

Next, we will show snippets (with omissions and translated into plain language) of 3 representative examples from the 205 MOPs.

#### 3.2.2.1 PE Router Software Upgrade

1. **Increase the OSPF weights** on the PE[3] router to 65535
2. Verify that traffic is drained
3. Shutdown BGP sessions to CE routers
4. Upgrade the drained PE router

[2]Health checks are performed after each step.
[3]Provider-Edge

5. **Reset the OSPF weights** on the PE router
6. Re-establish BGP sessions with CE routers

The above example shows how the operator drains traffic off a PE router before performing device upgrade, and "brings back" traffic after the change. This MOP is a perfect example of how minimal the input can be for a traffic migration job. Since 65535 is the maximum and default OSPF weight for drains, and software upgrade requires all links be drained, what operators need to provide as input for this traffic migration job is simply **which PE router**.

Note that in this example, to "bring back" traffic after the upgrade by resetting the OSPF weights, the operator needs to record the original OSPF weights. Additionally, this pattern of two opposite traffic migrations before and after a change is deployed is very common among the MOPs in our study. We will talk more about that in §3.2.5.

**Observations. 1.** The input of a traffic migration job can be as minimal as the identifier of the target. **2.** It is common to perform two opposite traffic migrations during an operation, which usually requires the operator to maintain certain state variables throughout the process.

#### 3.2.2.2 MME Software Upgrade

1. **Reduce "RelativeMmeCapacity"** of the target MME[4] to 0 so that incoming new connections will be redirected to other MMEs in the pool
2. Verify that number of connected UEs is below threshold
3. **Move connected UEs** to other MMEs in the pool
4. Delete static routes towards the isolated MME
5. Upgrade the isolated MME
6. **Reset "RelativeMmeCapacity"** to its original value

[4]Mobility Management Entity, an LTE control-plane function

7. Reinstall the static routes towards the target MME

In the above example, the operator needs to isolate an MME from its pool before the upgrade, then insert the MME back into its pool. Unlike routers, MMEs are **stateful** — the operator needs to migrate existing and redirect new traffic in order to isolate the MME from its pool.

The approach to preventing new connections on an MME is conceptually similar to that of a PE router: one lowers the capacity while the other raises the weight, both leading to a lower preferability. However, to migrate existing traffic, the operator has to (1) query the number of connected UEs on the MME and (2) explicitly move these UEs to other MMEs.

**Observations. 3.** Stateful network functions require two separate migrations for both new and existing traffic. **4.** Existing and new traffic are never distributed differently in any of the MOPs, and they are usually evenly distributed among peers.

#### 3.2.2.3 Network Zone Traffic Migration Through DNS Redirection

1. **Edit a list of DNS files** on DNS server "alnapnrdns01" to move traffic of Phone, Broadband, VoLTE, etc. to ALN[5] NZ[6]1, NZ3, NZ4, NZ5 and NZ6
2. **Edit a list of DNS files** on DNS server "alnapnrdns01" to move primary roaming traffic to ALN NZ3, NZ4

In this example, the operator draining an entire network zone 2 by migrating traffic of specific services to other network zones via DNS redirection. This MOP differs from the previous two in that operators are using DNS servers to steer traffic as opposed to directly reconfiguring target entities. Besides, for this operation, the operator is very specific about where each service traffic should go; the destinations of different service traffic cannot be inferred.

[5]Acronym of a location
[6]Network zone

Table 3.1: Key traffic migration parameters

| Parameter | Usage | Required? | Percentage |
|---|---|---|---|
| Target | The network entity around which traffic migration is performed. | Yes | 100% |
| Peer(s) | Network entities that receive or send traffic from or to the Target. | No | 23% |
| Weight(s) | In each migration, the distribution of traffic among the receiving entities | No | 22% |
| Filter(s) | Identifiers for any subset of the migrating traffic (*e.g.,* bit-masks) | No | 1% |

**Observations. 5.** Network function is not the only type of target for a traffic migration job. **6.** Traffic migration can happen on a more fine-grained level which involves a subset of the traffic of the target.

### 3.2.3 Key Parameters

We see from previous examples how simple traffic migration intentions get obscured by the detailed mechanics of specific procedures that vary drastically across network functions (NFs). To decouple the expression of intention and the actual execution details, we need to find a set of well-defined parameters that are generic yet carry enough information for specifics to be derived. Based on our analysis of the 205 MOPs, we identify four key parameters that satisfy this requirement, as shown in Table 3.1. To simply show their usage, here are how the three examples from §3.2.2 can be described using one or more of these parameters:

**Traffic Migration for PE Router Software Upgrade.**

1. Drain the traffic off {Target="Router-1"}
2. Bring back the traffic to {Target="Router-1"}

**Traffic Migration for MME Software Upgrade.**

1. Send traffic of {Target="MME-1"} to
   {Peers=["MME-2", "MME-3", ..., "MME-11"]} with
   {Weights=[10%, 10%, ..., 10%]
2. Bring back the traffic to {Target="MME-1"}

**Network Zone Traffic Migration through DNS Redirection.**

1. Send traffic from {Target="NZ2"} matching {Filters=["Phone","Broadband","VoLTE"]} to {Peers=["NZ1","NZ3","NZ4","NZ5","NZ6"]}
2. Send traffic from {Target="NZ2"} matching {Filters=["Roaming"]} to {Peers=["NZ3","NZ4"]}

### 3.2.4 Local and Remote Traffic Migrations

In previous examples, we can see that in some cases the operator performs traffic migration by directly reconfiguring the target; while in others, the operator uses an external component like a DNS server. We categorize these two methods as *local* and *remote* migrations, and call the external component an ***Anchor Point***. The difference between *local* and *remote* traffic migrations is whether it takes place on the *Target* or on *Anchor Points*.

### 3.2.5 Traffic Migration Complexities

As mentioned earlier, existing practices of traffic migration often couple operator's intention with network setups, which lead to complexities in the forms of additional information to acquire/maintain before/during each operation. In this section, we highlight these complexities identified in the MOP analysis.

**Migration Method & Anchor Point Discovery** For each traffic migration, operators need to determine what migration method the *Target* uses and identify the type and location of *Anchor Points* (if needed) based on their network setup (Figure 3.2).

**Target or Anchor Point Configuration** For both local and remote migrations, the diversity of *Targets* and *Anchor Points* translates unfortunately well to the diversity of their capabilities and how they are configured. *Targets* are not necessarily network functions, and stateful *Targets* incur additional procedures.

**Reverse Traffic Migration** As mentioned earlier, operators need to maintain state variables for reverse traffic migrations. Such externally maintained states not only incur addi-

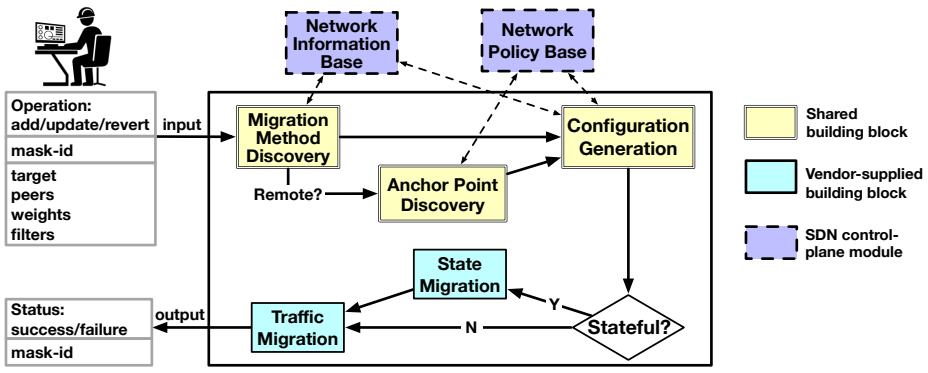



Figure 3.1: Egret's generic model and modular workflow

tional operational complexities, but they also leave operators susceptible to run-time failures.

**Parallel Job Coordination** Our MOPs suggest that operators always perform operations sequentially even when there are multiple devices in the same pool that need to be upgraded. This results in maintenance windows spanning hours if not days, which is highly inefficient. Manually performing parallel operations is error-prone and complex, especially when the same anchor points are shared.

## 3.3 Egret

From §3.2.3 we see that a few key parameters suffice to describe traffic migration intentions regardless of network function types and migration methods. This means that the amount of input required to carry out a traffic migration is minimal. Enlightened by this finding and observations on major traffic migration complexities (§3.2.5), we propose the design of Egret, a traffic migration system that (1) decouples high-level traffic migration intentions from low-level executions with a generic interface, and (2) modularizes low-level executions into functional building blocks that can be either shared or swapped in a plug-and-play fashion. In this section, we first present an overview of Egret's generic model and its modular workflow in §3.3.1, then we introduce an optimization Egret leverages to efficiently manage existing and in-coming traffic migration jobs in §3.3.2.

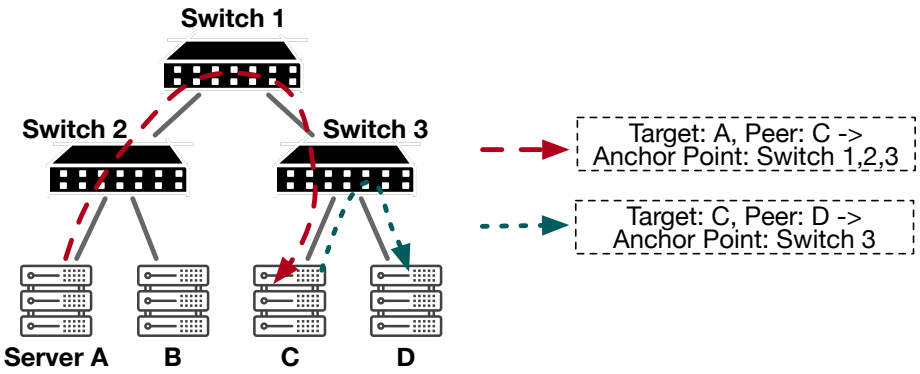



Figure 3.2: Anchor point discovery in a simple tree topology

### 3.3.1 A Unified Model and Building Blocks

Figure 3.1 shows an overview of Egret's design. Egret allows operators to specify their traffic migration intentions through a generic interface based on the key parameters described earlier: *Target*, *Peer(s)*, *Weight(s)* and *Filter(s)*. Some of the procedures discussed in §3.2.5, which contribute to most of the complexities of MOPs and now abstracted by Egret's interface, are modularized as building blocks in Egret's internal workflow (detailed below). Some of these building blocks are reusable, such as anchor point discovery and migration mechanism discovery, because they are universal and fundamental to all MOPs according to our study. Others are specific to network function types or anchor point types, such as state migration and traffic migration since they directly interface with underlying network functions. Vendors only need to develop these two building blocks in Egret's modular design, as opposed to writing completely new MOPs from scratch.

**Anchor Point Discovery.** As discussed in §3.2.5, traffic migration intentions are independent of the type and locations of anchor points. However, as shown by previous works [80, 81], the selection of anchor points does play an important role in traffic migrations in terms of performance(*e.g.,* packet loss) and service impact. Egret's anchor point discovery building block leverages network topology information stored in a network information base (NIB, commonly found in SDN control platforms) and follows operators' policy to find the most suitable anchor point(s) (if necessary). Note that there can be mul-

Table 3.2: Egret's mask-based APIs

| API | Description |
|---|---|
| add([params]) | add a new mask described by [params], require explicit removal without timeout |
| update(mask-id, [params]) | augment the [params] of an existing mask associated with the mask-id, including the *base* |
| revert(mask-id) | remove the mask associated with the mask-id |

tiple anchor points for one migration depending on the relative positions of the *Target* and *Peers(s)* in the network. Figure 3.2 shows a simple example where the servers' migration method is *remote* and their anchor points are switches.

**Migration Mechanism Discovery.** In §3.2.4 traffic migration methods are categorized as either *local* or *remote* based on where migration happens. After the discovery of *anchor points* from the topology, this building block determines the actual migration method the *Target* uses by querying the NIB.

**Configuration Generation.** This building block takes in parameters from user input and results of previous building blocks (anchor point and migration method discovery), and generates generic configurations for the corresponding network components. These configurations will then be consumed by vendor-supplied building blocks which translate them into device-specific commands. In this building block, a mask-like abstraction is used for efficient management of existing and in-coming traffic migration jobs (detailed in §3.3.2).

**Traffic & State Migration.** These two building blocks directly interface with *Targets* or *Anchor Points* to deploy the configuration changes for actual migrations. Compared with MOPs which are application-, software version- and network setup-specific, these two building blocks are, in the worst case, *Target*- or *Anchor Point*-specific. Leveraging generic configuration solutions like OpenConfig [82], these two building blocks can be further generalized and vendors' effort reduced.

### 3.3.2 Mask-Based Job Management

In §3.2.5 we mention that reverse traffic migration is one of the major sources of complexities in existing traffic migration workflows because it requires operators to (1) keep

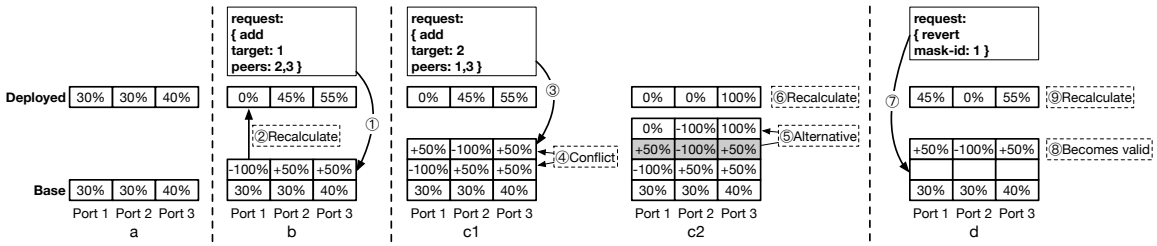



Figure 3.3: Job interleaving and reverse traffic migration: a 3-port load-balancer example

track of deployed configurations and (2) compute new configurations to revert existing ones. To minimize the amount of state operators have to maintain and enable easy reversions, we propose the design of a mask-based abstraction for traffic migration jobs that is easy to manage (add, update, revert). This design is inspired by configuration rollback capabilities commonly found in modern routers [83]. Egret's mask-based APIs are shown in Table 3.2.

Figure 3.3 shows an example of a 3-port load-balancer. Each port has a weight, indicating what portion of in-coming traffic should go out through each port. A mask (corresponding to one traffic migration request) consists of relative weights indicating the amount of traffic to be taken from or given to each port in a migration. Masks are put on top of one another into a stack, at the bottom of which is the *base* denoting the default configuration of the load balancer. The current (deployed) distribution is calculated by applying all masks to the *base*. In this example, the *base* is `{30%,30%,40%}`, indicating by default 30% of traffic goes out of port 1 and so on. In Figure 3.3-b a new traffic migration job comes in with the intention to migrate traffic from *Target* 1 to 2 and 3 (assuming *Targets* 1, 2 and 3 are connected to port 1, 2 and 3 respectively). This will generate a new mask `{-100%,+50%,+50%}` on top of the *base* resulting in a new configuration `{0%,45%,55%}`[7]. New masks always go to the top of the stack, while existing masks can be removed from anywhere. Whenever such a change happens, the current configuration is recalculated automatically.

[7]Because the 30% traffic of port 1 is distributed equally to port 2 and 3, each getting 15% of all in-coming traffic.

#### 3.3.2.1 Job Interleaving and Conflict Resolution

Through mask stacking, Egret allows different traffic migration jobs to interleave, meaning individual jobs can start and finish independently (assuming no conflicts exist between them). This is a significant improvement over traditional MOPs which are run sequentially (§3.2.5) and it is non-trivial to parallelize jobs when anchor points are shared. Egret achieves job interleaving by keeping track of individual masks and always recalculating actual configuration whenever changes happen. For example, in Figure 3.3-c1, a second job intending to migrate traffic from *Target* 2 to 1 and 3 arrives while job 1 is still active. The two jobs conflict with each other because *Target* 1 cannot be both source and sink. However, *Target* 3 is a sink in both requests, meaning an alternative exists for job 2. Policy-permitted, Egret can automatically resolve conflicts and generate alternative masks such as the one in Figure 3.3-c2. Additionally, in Figure 3.3-d, job 1 is reverted while job 2 is active. Since Egret always recalculates the configuration upon changes, the original mask 2 becomes valid and is applied to *base*. Job 1 and job 2 are never "aware" of each other throughout this process.

#### 3.3.2.2 Achieving Reverse Traffic Migration

In Figure 3.1, we see that Egret returns a mask-id with each execution. This allows operators to uniquely identify existing jobs. As mentioned in §3.2.5, reverse traffic migration brings an enormous amount of complexity because additional states need to be maintained and new configurations need to be generated. In Egret, with each mask kept track of individually, reverse traffic migration becomes as simple as reverting a mask, as shown in Figure 3.3-d.

#### 3.3.2.3 Updating the *Base* and Handling Run-time Failures

Sometimes the operator needs to re-adjust the default configuration of a network component, or more often a failure happens rendering the existing configuration obsolete

(*e.g.,* the number of available ports decreases because of a failed link). These situations can be easily handled by updating the *base*, either proactively by the operator or automatically through common SDN topology discovery mechanisms [29, 30, 9, 33].

## 3.4 Evaluation

In this section, we describe our prototype of Egret, and present both qualitative and quantitative evaluation results of Egret's simplification capability as a result of generality and automation.

### 3.4.1 Prototype

As shown in Figure 3.1, Egret consists of shared building blocks (generic functions that exist in all workflows) and vendor-supplied building blocks (device-specific functions that handle the communication between the control-plane and data-plane). We implement these building blocks as SDNator (Chapter V) applications: we build the shared building blocks from scratch and use existing control-plane frameworks (*e.g.,* Ryu [29]) as vendor-supplied building blocks by simply importing SDNator's DUE library. We also leverage SDNator's built-in databases to store and share network information such as topology and device configurations among building blocks. For example, after Ryu discovers all the OpenFlow switches and their link connections in the network, it can write this topology information to SDNator's Data Archives, which other building blocks (*e.g.,* anchor point discovery) can access when needed.

**Anchor Point Discovery.** For our prototype, we implement anchor point discovery using a simple algorithm that finds parent nodes of *Target* and *Peer(s)* and their closest common ancestor. This algorithm will allow Egret to correctly identify the anchor points in scenarios such as migrating traffic between servers on different racks, which involve ToR switches and aggregate switches (Figure 3.2). One can easily upgrade to a more sophisticated algorithm [84] that selects the best anchor point(s) among several candidates based

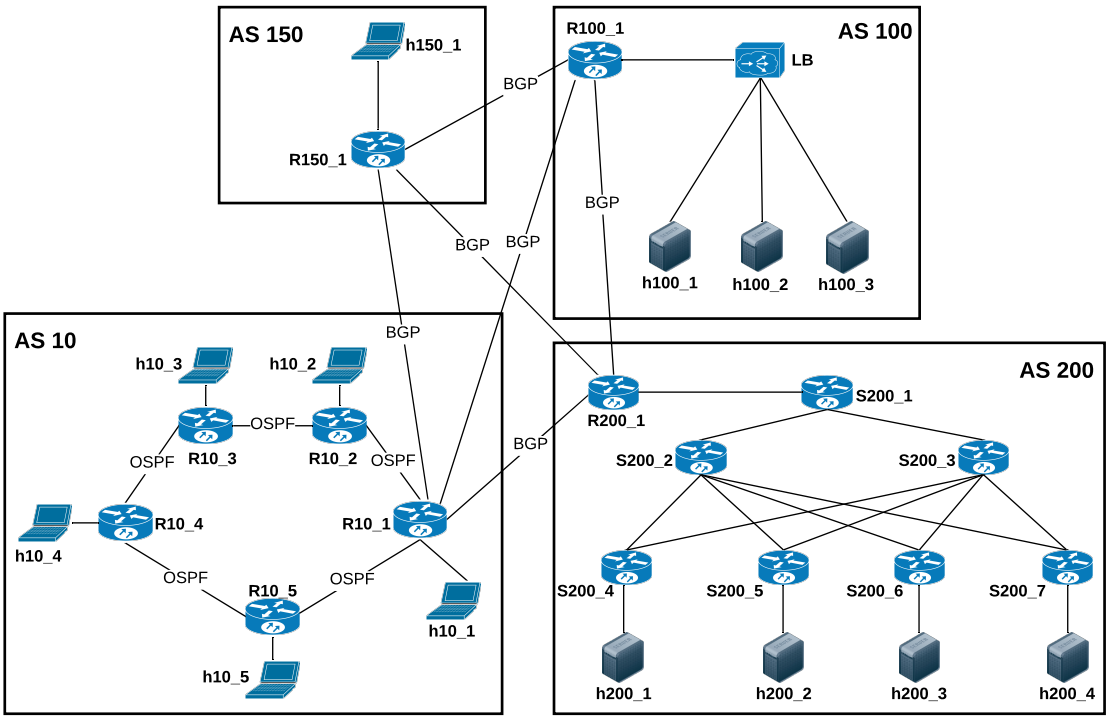



Figure 3.4: Emulated network topology with four ASes

on global traffic dynamics and operator's policies with Egret's modular design.

### 3.4.2 Egret in Real-World Workflows

To evaluate how Egret handles heterogeneous devices in practice, we test it in three different scenarios (as detailed below) that involve both local and remote traffic migrations (§3.2.4), and all four parameters (§3.3.1. Figure 3.4 shows the network topology for this experiment. We use Mininet [85] to create a multi-AS internet running on BGP. ASes 10, 100, and 200 have different network setups to showcase the different scenarios: AS 10 consists of Quagga [86] routers running on OSPF for intra-domain routing; AS 100 features an NGINX [87] load-balancer and three HTTP servers; AS 200 emulates a fat-tree [88] topology with programmable switches. In each scenario, we integrate Egret in a typical workflow where we perform traffic migrations using Egret's unified interface (§3.3).

#### 3.4.2.1 Router Upgrade

The first scenario relates to the router upgrade example (traffic migration part) in §3.2.2.1, where *Target* is the only input and traffic migration happens locally via OSPF link weight adjustments. As shown in Figure 3.4, AS 10 consists of five OSPF routers that provide two paths between host `h10_4` and `h10_1`, one via router `R10_3` (5 hops) and one via `R10_5` (4 hops). Initially, we configure all OSPF weights to be equal, which means the lower path via `R10_5` is the favored path with a smaller cost. We then send a traffic migration request to Egret with `R10_5` as the *Target*, which will result in its OSPF weights being raised to the maximum, effectively blocking all future traffic (after OSPF protocol converges). After a certain period (*i.e.,* router upgrade finishes), we call Egret's API to remove the previous mask by simply providing the `mask-id`, which will bring back traffic from the longer path via `R10_3`. Throughout this process, host `h10_4` acts as the traffic source and keeps an active connection with `h10_1`. For this scenario and the following ones, we trigger the traffic migration event after 20s and the reverse traffic migration after 40s. The results are shown in Figure 3.5.

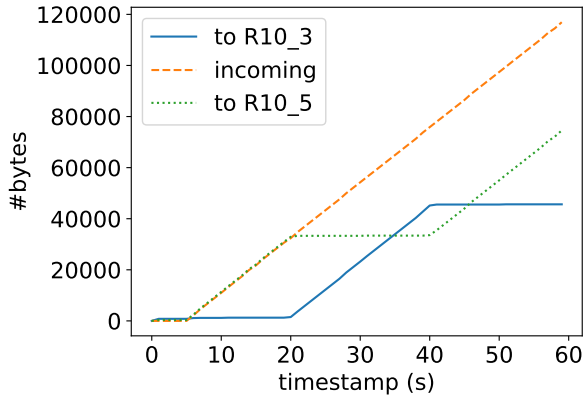



Figure 3.5: Router R10_4 traffic pattern before and after R10_5 upgrade

#### 3.4.2.2 HTTP Server Load Balancing

The second scenario resembles the 3-port load-balancer example described in §3.3.2. In this scenario, traffic migration happens with an anchor point, and Egret takes in more

parameters (*Target*, *Peers*, and *Weights*). As shown in Figure 3.4, AS 100 consists of one L4 load-balancer and three HTTP servers. The load-balancer uses a round-robin approach to evenly distribute new TCP connections to each server. We use host `h150_1` in AS 150 as the traffic source to initiate HTTP requests. At tick 20, we send a traffic migration request to Egret with server `h100_1` as the *Target*. Unlike in the previous scenario, where the OSPF protocol automatically handles the traffic shift, we need to explicitly instruct the load-balancer to adjust the traffic distribution. In this case, we set the *Peers* to be `h100_2` and `h100_3`, and the *Weights* to be `{-100%,+50%,+50%}`, which will drain `h100_1` and evenly redistribute its traffic to its other peers. At tick 40, we remove the mask as in the previous scenario to bring back traffic to `h100_1`. Results are shown in Figure 3.6.

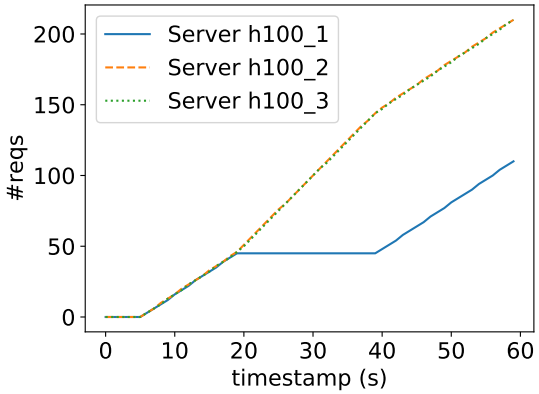
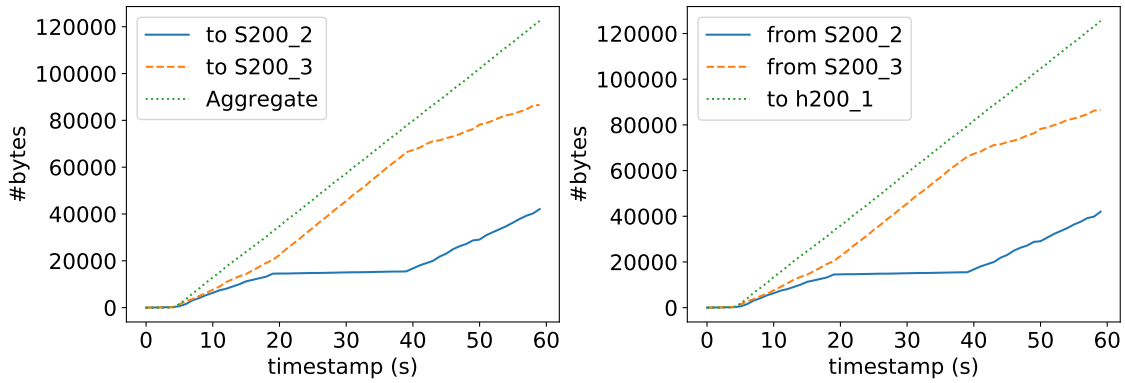



Figure 3.6: Workload distribution before and after server load balancing

#### 3.4.2.3 Switch Failure Mitigation

The previous two scenarios showcase Egret's capability in handling proactive traffic migration requests for heterogeneous workflows. What happens if there is a failure and requires reactive traffic migration to avoid packet loss? We mention in §3.3.2 that Egret's mask-based approach can be handy for dealing with run-time failures. To demonstrate this capability, we create a fat-tree topology with OpenFlow switches in the third scenario, as shown in Figure 3.4. We use Ryu as the standard SDN controller that actively keeps track of the network topology in the control-plane, and leverage SDNator to share that

information with Egret in real time. We use `h150_1` in AS 150 as the traffic source to continuously send packets to server `h200_1` in AS 200. All switches operate under an equal-weight setting that distributes the traffic fairly evenly among its peers, which creates two paths leading to `h200_1`: `[S200_1->S200_2->S200_4 ->h200_1]` and `[S200_1->S200_3->S200_4->h200_1]`. We manually disconnect `S200_2` from the network to simulate a failure, which is immediately detected by Ryu and leads to Egret changing the *base* mask for `S200_1`. A change in *base* will result in `S200_1` being reconfigured to only send packets to `S200_3`. Later we restore `S200_2` and submit a permanent (*i.e.,* will update the *base*) traffic migration request to Egret to restore the original traffic distribution. Results are shown in Figure 3.7.




(a) S200_1 (b) S200_4

Figure 3.7: Traffic pattern before and after S200_2 failure

### 3.4.3 With and Without Egret

By generalizing traffic migration with its model and decoupling intention specification from executions for automation, Egret can significantly reduce the complexities for traffic migration compared to traditional MOPs. One reduction is in the number of operations performed by the operator. As mentioned earlier, traditional MOPs require operators to run low-level configuration commands or modify configuration files line by line (*e.g.,* §3.2.2.3), which is error-prone and time-consuming, especially when the number of operations increases with the size of the network. Egret reduces this complexity by allowing operators

Table 3.3: Comparison in operational complexity with and without Egret

| | # of operations | | # of state variables | |
|---|---|---|---|---|
| | MOP | Egret | MOP | Egret (+1 for mask-id) |
| Router | 5-10 | 2 | # of links | 1+1 |
| MME | >40 | 2 | >10 | 3+1 |
| DNS | >100 | 1 | >50 | 3+1 |

to express their intentions using the mask-based API with a few key parameters. As shown in Table 3.3, Egret consumes only two API calls for jobs that require reverse traffic migration (add and revert mask respectively). Another significant reduction in complexity is the number of state variables that need to be acquired and maintained during traffic migrations. Using Egret, operators only need to know the *Target* in the router example; *Target*, *Peers* and *Weights* in the MME example; and *Target*, *Peers* and *Filters* in the APN-DNS example. And the only state that is maintained is mask-id. Whereas in MOPs, operators need to record existing configurations and states (link OSPF weights, MME connection stats, DNS entries, etc.) while running tens or hundreds of lines of commands.

## 3.5 Discussion

**Egret's applicability to all traffic migration applications and network function types.** Even though Egret applies to the 205 MOPs from our study, it is impossible to claim universal applicability. We do, however, make a few design choices to future-proof Egret: (1) the *Filter* parameter of Egret's API supports any packet matching patterns in the form of bit-masks; (2) Egret's categorization of migration methods is very coarse-grained: *local* and *remote*, which makes it easy for vendors to plug-and-play building blocks.

**Conflict detection with Egret's mask-based approach.** As mentioned earlier (§3.3.2), Egret rejects jobs that use existing traffic sources as sinks and vice versa. This is intuitive for jobs that intend to completely drain their *Targets*. For jobs that migrate traffic partially, however, this policy can be too strict. Exploring more flexible policies would

be an interesting next step.

**Time reduction by Egret.** With automation and job parallelization, there is likely a considerable amount of time reduction when using Egret as opposed to MOPs. It is, however, non-trivial to quantify this reduction since (1) traffic migration is one of many procedures in a MOP and (2) each traffic migration is device-, traffic- and policy-dependent.

## 3.6 Related Work

Traffic management, especially in the context of SDN, is not a new topic. Previous works have provided many instrumental tools — scalable and efficient rule management in control and data-plane [89, 90, 91], abstractions for forwarding elements [92, 93, 18], network function state management [42, 94, 95], and traffic engineering [96, 97]. Egret's ultimate goal is to be able to incorporate and leverage all these tools for different kinds of applications in different network setups, while maintaining a unified interface.

Maelstrom [79] is a traffic management framework that shares some insights with Egret such as abstracting traffic migration intricacies with generic interfaces and providing reusable primitives to compose workflows. However, Maelstrom is tailored to traffic management for disaster mitigation and recovery and is based on a load-balancer-only setup. Egret is a more general approach that abstracts traffic migration from different applications and network setups. Particularly, Egret addresses the heterogeneity of network functions, which is less prominent in data-center settings such as maelstrom's. Maelstrom has also addressed several technical challenges for traffic migration such as preserving service dependencies, which can be incorporated in Egret's building blocks.

Load balancing is a common way of traffic migration. Previous works either use software load-balancers [98, 99, 100, 101, 102] or propose new solutions [103, 104, 105] for traffic shifting. While these works focus on improving load-balancing, Egret focuses on abstracting traffic migration, with load-balancing being one of the *remote* migration methods.

Network function state management is closely related to traffic migration. OpenNF [42], S6 [95] and OFM [94] all use an SDN-based mechanism (SDN switch) to steer traffic which is another way of *remote* migration in Egret's model. Khalid et al. [106] proposed standardized APIs for VNF management but didn't focus on generalizing and simplifying traffic migration as Egret does.

## 3.7 Summary

We revisit traffic migration, a common procedure in many network operations, in the light of rapidly emerging virtualized network functions that leads to increasingly *heterogeneous* networks. Our analysis of 205 change management Methods of Procedure (MOPs) sheds light on the commonalities and complexities of existing traffic migration practices, which motivate the design of Egret. Compared to traditional MOPs, Egret can significantly simplify traffic migration through abstraction (with a unified model and generic APIs), modularization, and efficient state management (with mask-based abstractions).

# CHAPTER IV

# ADD: Application and Data-Driven Controller Design

In this and the next chapter, we turn our attention to *data-driven* applications, which are extremely common in cyber-physical systems and increasingly important in network systems. For this chapter, we focus on studying the unique characteristics and requirements of data-driven applications and identifying the deficiencies of existing SDN systems in supporting data-driven applications. Based on the findings of our study, we propose a data-driven controller design, which enables applications to gather more information from the data-plane and make more informed decisions faster.

## 4.1 Introduction

The rise of SDN not only marks the transition of a fully-distributed network architecture to a hybrid one with logically centralized control, it also puts a strong emphasis on the roles applications can play in network management and even design. One of the many outstanding features offered by SDN is a global view of the network, which empowers applications with greater freedom and visibility to programmatically manage the network. To some extent, the more information the controller is able to provide to applications, the more capable applications are in gaining insights of the underlying data plane and making (globally) optimal decisions.

**Existing SDN controllers are mostly event-driven.** In existing SDN controller im-

plementations [29, 30, 47], applications mostly rely on network events such as *link down*, *link congested*, *new host up* and *no matching rules found for packet* to trigger reactive procedures like rerouting.

**Systematic support for data access is limited.** Various existing southbound protocols [107, 12, 1, 17] do allow SDN applications to proactively pull data and states such as *port/link stats*, *packet headers*, and even *full packets* from data plane components to gain more insights. However, lack of systematic support for such operations means application development is tightly coupled with specific southbound protocols. Moreover, each application acting independently leads to redundant data retrieval across different applications due to lack of request consolidation, and duplicate data copy residing in each application.

**Future SDN applications will be more data-driven.** As network composition continues to diversify in terms of network function types (NFV) and end-host types (IoT, autonomous vehicles, VR/AR devices, *etc.*), the need for network service (*i.e.,* features offered by SDN applications) customization and flexibility will increase dramatically, so does the volume and diversity of data produced on the data plane. Different end-hosts might have drastically different requirements for network services in terms of QoS guarantees, policies, and even intensive data analysis. To satisfy these requirements, SDN applications cannot simply rely on existing network events and data digests that are usually device-agnostic and contain scarce information. Instead, they must have direct, flexible and efficient data access, as should also be provided by the controller.

Unfortunately, as we discussed earlier, existing SDN controller implementations commonly adopt an event-driven model that lack general support for applications to read data freely and efficiently. Therefore, in this work, we present ADD, a new application and data-driven SDN controller design that aims to improve data access flexibility and efficiency for SDN applications. Several design choices are made in ADD to satisfy and resolve the aforementioned requirements and limitations: (1) Applications subscribe to data in addition to events; (2) Data subscriptions are consolidated to eliminate redundancies; (3) Southbound

interface uses generic key-value based schema; (4) Northbound interface is divided into high-level intent-based APIs and low-level APIs.

ADD's design complements instead of contradicts existing event-driven designs. Although we focus on the data-driven aspect in this work, we believe both event-driven and data-driven models are needed for different applications.

We make the following contributions in this work:

- We identify the gaps between existing SDN controller design and the requirements of data-driven applications. We carry out a case study on smart manufacturing systems and applications to understand the implications of data-driven applications.
- We propose an Application and Data-Driven (ADD) controller design that provides general, flexible and efficient data access for applications, and generic interfaces for controller and applications to process data.
- We prototype ADD and present preliminary evaluation results that show the scalability, performance and usefulness of the new design. We develop two applications using the new model to gain more insights of the data plane and outperform existing solutions.

## 4.2 Case Study

In order to gain a better understanding of data-driven applications, especially the volume and characteristics of data they deal with, and their requirements for the controller, we carry out a case study on smart manufacturing systems [26] and applications with an actual smart manufacturing testbed (Figure 4.2). Smart manufacturing systems are data-driven and heterogeneous in nature, which share many commonalities with a data-driven SDN. Table 4.1 shows the comparisons between both types of systems on their data/production plane composition, applications that they run, and the characteristics of data they produce and consume.

Table 4.1: Comparisons between SDN and Smart Manufacturing Systems

| | **Legacy SDN** | **Smart Manufacturing Systems** | **Future Data-Driven SDN** |
|---|---|---|---|
| Data/Production Plane Composition | Programmable Switches | Conveyor belts, Robots, CNCs, Motors, Printers, *etc.* | Programmable Switches, Physical and Virtualized Network Functions |
| Applications/Tasks | 1. Routing<br>2. Firewall<br>3. Traffic Engineering<br>4. Network diagnostics *etc.* | 1. Routing<br>2. Predictive Maintenance Control<br>3. Scheduling<br>4. Anomaly Detection *etc.* | 1. All legacy SDN apps but more complex<br>2. Device-specific service customization (Phones, Vehicles, wearables, sensors, *etc.*)<br>3. Emerging apps like VR, AR, *etc.*<br>4. Real-time network telemetry |
| Data/States | Mostly discrete, real-time | Both continuous and discrete, real-time and historical | |

**Networking v.s. Manufacturing.** Despite being rather different domains, networking and manufacturing share quite some similarities. For instance, consider *Routing* in networking where a network path is selected to forward data packets: the manufacturing equivalence would be to select a physical path to transport materials or parts. Both applications need visibility of data/production plane's topology and diverse properties such as device capabilities, bandwidth, and queue length. As shown in Table 4.1, legacy SDN has a relatively homogeneous data plane with mostly programmable switches, while manufacturing production plane and future SDN data plane are much more heterogeneous. On a manufacturing production plane, each machine could process parts with different structures and generate a huge amount of data due to continuous physical properties like voltage, current, *etc.*. Manufacturing applications utilize these data to make real-time decisions, monitor machine status, and predict failures. Similarly, P4 [17] allows packet formats to be customized and different VNFs generate a wide spectrum of different data and states. Historical data also plays a significant role in manufacturing applications. If SDN applications can make use of all these data across different sources, they can have a much better visibility of the network and deliver features that were not possible before.

**Insights.** Through this case study we intend to explore the data-driven aspect of future SDN. These comparisons give us a fresh perspective of how SDN composition might look like in near future and how data could become a dominant factor in future application development. We envision future SDN to benefit even more from such a data-driven

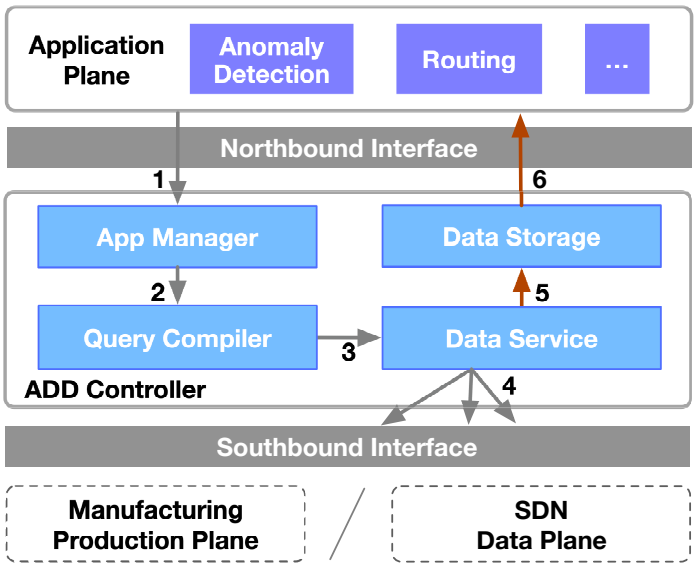



Figure 4.1: Data flow of the ADD controller design

controller design because of the diversifying end host types (phones, vehicles, sensors, wearables, *etc.*) in addition to the already heterogeneous network functions.

## 4.3 ADD Controller Design

Figure 4.1 shows our proposed ADD controller design. In brief, ADD decouples applications from data retrieval and storage and provides general data access support with its internal pipeline and generic interfaces (described below). As a result, application developers can solely focus on "what data to request" and "what to do with it".

### 4.3.1 Key Controller Modules

The controller has four main modules that provide applications with efficient data access capabilities. As shown in Figure 4.1, (1) Apps register themselves to the App Manager with their interests (i.e. the data they want); (2) Query Compiler traverses all the interests, converts them into corresponding data fields, and consolidates all fields as a minimal set of queries and (3) passes them to the Data Service; (4) Data Service continuously queries data (a subset as defined by the interests) via the southbound interface and caches them in

the Data Storage; (5) Applications fetch data from the Data Storage which contains both raw data (as explicitly requested by the applications) and network states (*e.g.,* global view). Both historical data and real-time data are stored in the Data Storage. Data storage supports both data-driven and event-driven models: applications can directly read data from it, or listen to changes of network states.

Applications' fetching data is asynchronous to controller modules' operations. This publish-subscribe-like messaging mechanism enables on-demand data extraction. Specifically, at the time data is produced or consumed, the producer and the consumer does not need to communicate with each other.

### 4.3.2 Southbound Interface

The southbound interface has to satisfy requirements of high scalability and low latency. As the number of applications grows, and with the increasing data volume they read from the production plane, the workload of the southbound interface rises significantly. Moreover, as discussed in §4.2, both manufacturing production plane and future SDN data plane will be heterogeneous. A generic abstraction is desired in order for the controller modules and applications to evolve/function independently from different devices.

To support abstractions and to cope with the potentially large data volume and the demands for structured, contextualized data from upper layers, we propose a customizable information model to describe the production/data plane. Specifically, data items (essentially key-value pairs) are grouped into objects which reflect device properties and functionalities. Each object can have multiple instances. For example, a CNC spindle has four axes, therefore the `Axis` object have four instances `Axis_X`, `Axis_Y`, `Axis_Z`, and `Axis_S`. The `Axis` object consists of spindle axis properties, such as `speed`, `acceleration`, and `voltage`. If certain data items in an instance are not of interest to any running applications, the southbound never queries them for better efficiency.

### 4.3.3 Northbound Interface

ADD provides flexible and efficient RESTful APIs for applications to communicate with the controller. As mentioned in §4.1, future applications will require frequent, sizable data exchanges with the controller, and these applications will need much more computing resources to perform data analytics and machine learning algorithms [108] which can easily overwhelm the controller. It is also challenging to manage application life-cycles, security and privacy [109] when they are co-located with the controller. Therefore, we envision applications to be moving out of the controller and be remotely communicating with it. The trend of geo-distributed computing [110, 111] and the aforementioned concerns make us believe that a bandwidth-efficient and low-latency northbound interface is desired. Existing RESTful northbound APIs [29, 30, 46, 47] in SDN controllers have two major limitations: (1) Data is organized in a coarse-grained manner thus client could get more than they want while bandwidth is wasted; (2) Client may have to submit multiple requests that correspond to different URLs to get all its desired data, which causes significant delay (although HTTP pipe-lining can help mitigate this problem, it can cause other problems such as Head-Of-Line blocking [112]).

ADD maintains a hierarchical data structure that houses fine-grained data fields and allows declarative query [113], in order to address the aforementioned issues. This data structure defines the capabilities (*e.g.,* available data items) of the northbound interface. Both the server-side (controller-side) northbound service and the client-side (application-side) northbound libraries have visibility of this definition. The client makes a single query to get all its desired data by strictly following the definition, *e.g.,* specifying a set of data items associated with a set of data plane components. Data service in the controller refers to it to execute queries from clients and only returns the exact data that clients need, nothing more, nothing less.

To offer great usability, we design two types of APIs, *i.e.,* intent-level APIs and task-level APIs. Hierarchical northbound design [18] has been proposed to provide better appli-

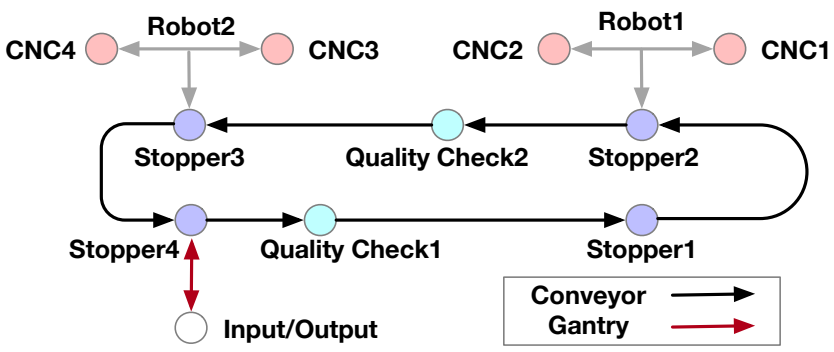



Figure 4.2: Physical layout of the testbed

cation programming experience in event-driven SDN. We believe it is also helpful in our data-driven approach. Task-level APIs perform specific tasks such as querying a given set of data items, computing reconfigurations, *etc.*.

## 4.4 Prototype and Evaluations

To understand how ADD performs in practice, we build a prototype ADD controller (as described in §4.4.1) using open-source software. We also develop two applications (§4.4.3): *anomaly detection* and *routing*, which exist in both manufacturing and networking domains. In this section, we first introduce our prototype ADD controller and discuss some of the implementation choices, then present both quantitative and qualitative evaluation results. Specifically, we evaluate the ADD design in terms of scalability, usability and performance. We carry out most of our experiments on an actual manufacturing testbed in proximity with our controller. As shown in Figure 4.2, the testbed consists of 4 Computer Numeric Control milling machines (CNCs), 2 Robots, 2 quality check modules and a conveyor loop. As parts travel on the conveyor, they might encounter machines with different functions. For example, when a stopper pauses a part, a robot can pick it up and places it on either one of the two CNCs depending on the part model. We can clearly see the analogies between this testbed and an SDN data plane.

Table 4.2: *Intent-level* and *task-level* northbound API examples

| API | Arguments | Description |
|---|---|---|
| **findRoute()** | device_id, part_id | Find an alternative route for a set of parts. |
| getTopology() | status_type | Return a graph representation of the topology with associated device status. |
| isConnected() | initial_id, candidate_id | Check whether the initial device and candidate device is connected. |
| hasCapability() | device_id, part_id | Check whether a specific machine is capable of processing a set of parts. |
| isReady() | device_id, part_id | Check whether a device is running the program that aims to process a part. |
| **detectAnomaly()** | anomaly_type | Detect anomalies within a specific type. |
| getDevice() | device_id | Get the statistics of a set of specific machines. |
| isFailed() | device_id | Check whether a specific machine is failed. |

### 4.4.1 Prototype

**Controller.** Existing SDN controllers such as Floodlight [30] adopt a publish/subscribe model for applications to choose which events to listen to. ADD adopts a similar mechanism but with one major modification: instead of the controller dispatching data to applications, applications proactively pull data from the data storage, with the exception of events (as detailed below). Because of the heterogeneity of devices and the unstructured nature of their data, we implement the data storage with MongoDB [114]. The data storage stores not only the data read from southbound but also the global view, which reflects the dynamics of the physical layout of production plane. The global view provides the connectivity of manufacturing units to applications like *routing* which require machine capabilities and the available paths between them. The initial layout of the production plane is constructed from a pre-defined configuration that describes the interactions and paths between units. This global view updates automatically based on the streaming data obtained from southbound. Instead of waiting for applications to pull the data storage and potentially discover this event, the controller notifies interested applications. This shows how event-driven model and data-driven model can work together to better serve application needs.

**Southbound.** OPC [115] is the most widely deployed standard for data access in industrial automation. In OPC, all the data items are provided as *tags*. Hardware vendors define default tags that their devices produce. Meanwhile, operators can define specific input and output bits, internal temporary variables as tags. Instead of reading from or writing to I/O bits, machine controllers are able to use tag names in their code to easily read and write

data. Our southbound interface implementation leverages OPC to get specific data tags from individual machines. Additionally, we define the information model using Protocol Buffers [116]. The benefits of the information model are twofold. First, it assembles scattered tags into meaningful data structures to describe machine status. Second, it provides an abstraction so that the controller can be completely agnostic to production plane details, which means the production plane (or data plane) can be swapped without changing the fundamental design of the controller. Both real-time data and historical data of the production plane are provided, and the controller can be deployed anywhere and retrieve data through remote procedure calls (RPCs) based on gRPC.

**Northbound.** We build our northbound interfaces on top of GraphQL [117] which provides a type system that is an ideal vantage point for us to define hierarchical data mappings. The GraphQL server directly interacts with MongoDB in the controller, while the GraphQL client resides in the app-side northbound libraries. We implement six task-level APIs and two intent-level APIs as shown in Table 4.2. `findRoute()` is an intent-level API that consists of four task-level APIs, which are `getTopology()`, `isConnected()`, `hasCapability()`, and `isReady()`. The other intent-level API, `detectAnomaly()`, is composed of two task-level APIs `getDevice()` and `isFailed()`.

### 4.4.2 Microbenchmarks

**Southbound Performance.** We read different numbers of data items from the production plane in order to measure southbound latency. Figure 4.3 shows the average reading time from 1 item to 200 items in 50 measurements. Overall, the latency is proportional to the number of data items being read, but there are some spikes of the average read latency. We find that those spikes come from outliers caused by unusually long OPC reading. Although there is no similar southbound implementation for comparison, the average latency of reading 200 tags is no worse than an industrial data collection tool named Rockwell

Cloud Agent Elastic.

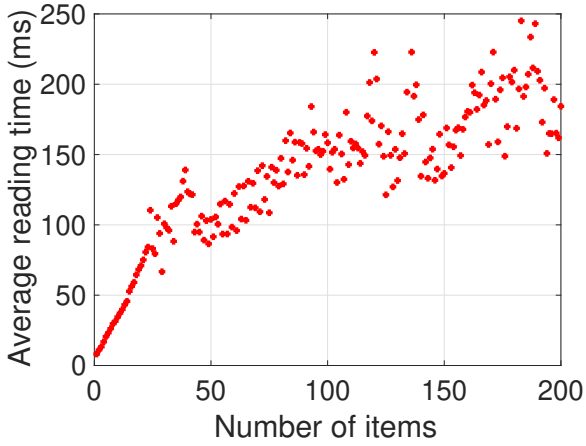



Figure 4.3: Southbound average reading time

**Controller Scalability.** Because of SDN controllers' centralized design, scalability is one of the most important criteria when evaluating their feasibility. For ADD's data-driven design, scalability is especially critical. For example, the overhead of controller writing data in the data storage should be contained despite large number of applications and interests. In order to evaluate this, we measure the time between a CNC going down on the production plane and the global view being updated in the data storage when varying the number of interests. If this time interval doesn't go up with the number of interests, it means (1) ADD's data storage has a stable overhead; (2) even with a large number of applications and interests, global view is updated fairly quickly.

The CNC we experiment with has a total of 800 tags, and we vary the number of tags queried by the controller from 15 to 800. The average time intervals measured are $9,203$ms, $9,276$ms, $11,278$ms respectively for 15, 100 and 800 such tags. This means with more than 50X the workload, the overhead increases for merely 22%. Note that since applications may query overlapping data items, ADD's data consolidation feature helps further improve the scalability. Besides, here we only use a single controller instance with a single southbound channel. In the future we plan to experiment with a larger scale of applications and data plane components, and better understand ADD's scalability by distributing the

controller instances and parallelizing the streaming channels.

**Northbound API Usability.** To evaluate the ease of use of our northbound API, we develop two first-of-its-kind cyber manufacturing applications: anomaly detection app and routing app. The total LoC of the anomaly detection app and routing app is 78 and 210, respectively. More detailed evaluation of the application will be introduced in §4.4.3.

**Northbound Performance.** To evaluate bandwidth efficiency and latency performance of our northbound interface, we run a benchmark app on a laptop that is one hop (WiFi link) away from the controller. Our app monitors the TX bytes and the current bitrate of the first port of two switches. We compare with RESTful APIs in Ryu [29] which provides two related RESTful APIs for querying these two statistics. They are (1) `GET:/stats/port/<dpid>[/<port>]` and (2) `GET:/stats/portdesc/<dpid>/[<port>]`. For comparison, we implement both of them in our controller. We assume the controller already has the data stored in its database and study the performance of the northbound interface itself. We measure the amount of data exchanged and the response latency for getting a pair of bitrate and TX bytes values. ADD's northbound interface consumes 255 bytes of data over the network while Ryu consumes 1,516 bytes. This disparity is attributed to the fact that ADD gets the exact data the app wants while Ryu delivers all the related data and lets the app do the data retrieval. The latency is 9.1ms and 43.4ms for ADD and Ryu, respectively. The improvement is brought by using just one query to fetch all the desired data, unlike Ryu where multiple requests have to be made to separately fetch data from different URLs.

### 4.4.3 System-Level Test With Applications

We develop two example applications (co-located with the controller) to test if ADD can (1) provide data access with low latency and (2) offer great programmability to applications so that they can benefit from the extra data access.

**Anomaly detection.** To showcase the capabilities of ADD and the low latency of the southbound and the northbound interfaces, we develop an anomaly detection application that looks into network traffic and machine status simultaneously to detect anomalies and diagnose provenance. We are able to detect anomalies earlier than tools that rely purely on machine status. We compare our anomaly detection application with the Rockwell FactoryTalk program. While the testbed is running, we inject anomalies into CNC3 and measure the the detection delays of our application and FactoryTalk [118]. Our anomaly detection application reports an anomaly after 12.6 seconds, while FactoryTalk detects that more than 35 seconds later. The root cause is that the underlying OPC has a caching mechanism, but we are able to tell anomalies sooner with data collected from network communications. Once an anomaly is detected, we let the anomaly detection application notify the routing application for computing a new route and deploying the reconfigurations.

**Routing.** The routing app awaits routing requests coming from the controller or other apps such as anomaly detection and machine maintenance. When the routing app receives notifications of a failure and that a part needs to be rerouted, it analyzes the topology and capabilities of each machine to make the rerouting decision. Our key metric is the application response time – the time between the reception of notification and when the rerouting decision is made. We repeat the experiment for 10 times and show the average values. It takes 10.9ms to query machine and topology data from the controller, 2.9ms to analyze the available machines, 1.7ms to check the capable machines, and 1.4ms to find the ready machines. The total application response time is 16.9ms.

**Interactions between the two applications.** With both apps installed, the administrator can leverage them to quickly detect failure and make rerouting decisions. In our evaluation, the routing app registers to listen for notifications from the anomaly detection app described above. Summing the response time of the two apps results in a 13s latency. Compared to traditional manufacturing system that usually takes hours, ADD significantly reduces the time it takes to (1) detect an anomaly and (2) deploy a new configuration.

## 4.5 Related Work

**Legacy SDN Controllers.** Existing SDN controllers [29, 30, 47] have a common event-driven model without general data access support (usually offered by southbound protocols like OpenFlow or NetFlow). ADD provides systematic data access support for SDN applications and adopts a data-driven model that decouples applications from data retrieval and storage. ADD complements rather than contradicts existing controller designs in that one could still use protocols like OpenFlow or P4 for reconfigurations of certain devices.

**Data-driven Approaches.** The integration of big data or data analysis with SDN is becoming an interesting topic for both areas. Some architectures are proposed [119, 120, 121, 122] to facilitate data analysis in SDN. These controller designs either focus on data analysis on the application side without providing systematic support for efficient data query and storage, or do not address the limitations of existing proprietary southbound interfaces that support limited data access. Other works like Marple [123] and Sonata [19] improve the flexibility and scalability of network monitoring/telemetry by leveraging state-of-the-art programmable switches. ADD is a generic controller design that interfaces with different kinds of applications and data plane devices.

**Hierarchical Controller Design.** Previous works such as SoftMoW [62] propose hierarchical structure to improve SDN controller scalability. Such effort is orthogonal to our work and could potentially be utilized to further improve our controller design (*e.g.,* improving southbound scalability by having multiple children controller instances).

## 4.6 Summary

We envision future network applications to be more data-driven as a result of increasing device heterogeneity and service customization. In this work, we carry out a case study of smart manufacturing systems and applications to (1) understand their unique data-driven

characteristics and requirements, and (2) identify the gaps between existing SDN controller designs and these requirements. This motivates us to propose an Application and Data-Driven controller design, or ADD, that aims to improve data availability for future SDN applications. ADD adopts a data-driven model that decouples applications from data retrieval and storage, and allows applications to gain more insights of underlying devices. We prototype ADD and evaluate it with two example applications that show the scalability and usability of the ADD design.

# CHAPTER V

# SDNator: Enabling Extensible Data-Driven Control in Cyber-Physical Systems

In the previous chapter, we study the unique characteristics and requirements of data-driven applications, and propose design improvements towards a data-driven controller with better data availability for emerging applications. Inspired by these insights, in this chapter, we describe the design and implementation of an extensible framework called SDNator for building centralized controllers for applications in network and cyber-physical systems. Despite its simplistic design, SDNator addresses several critical challenges with enabling centralized control in cyber-physical systems, and delivers high scalability and performance. With SDNator, we implement the first digital-twin-equipped central controller for additive manufacturing fleets, and show that SDNator-based centralized solutions significantly outperform distributed approaches in shortening production time, mitigating failures and anomalies, and optimizing production plans upon urgent requests such as producing Personal Protective Equipment (PPE) during the COVID-19 pandemic.

## 5.1 Introduction

Recent advances in hardware capabilities (*e.g.,* programmable switches, autonomous automobile, smart manufacturing) and network connectivity technologies (*e.g.,* 4G and 5G)

have enabled new software-hardware interactions in cyber-physical systems (CPS). For instance, a centralized control architecture inspired by Software-Defined Networks (SDN) has been widely explored as a viable alternative to traditional distributed solutions in various CPS such as manufacturing [26, 28, 27], IoT [23], and autonomous driving [25, 24]. Applications can utilize aggregated real-time information of a CPS to improve its flexibility, efficiency and reliability. Furthermore, this paradigm opens up exciting research opportunities for new control strategies and workflows that were previously infeasible.

However, developing and deploying CPS applications based on this centralized paradigm is non-trivial. Due to a lack of available "controller" frameworks that readily support CPS applications, existing works often resort to extension or adaptation of legacy SDN controllers to tailor to their particular application requirements [26, 23, 124]. Such extension or adaptation is time-consuming, non-reusable, and inherits the same limitation in supporting data-driven CPS applications (*i.e.,* inefficient data production/consumption, lack of data heterogeneity and historical data support, *etc.* [36]). If a new "controller" variant has to be built every time new CPS applications emerge, which is likely given how heterogeneous CPS are, the value of flexibility and programmability diminishes.

To fundamentally address this problem, we design, implement and open-source SDNator (§5.2), a framework that enables researchers and developers to easily implement new or incorporate existing CPS applications in a centralized workflow. SDNator is not a specialized controller; rather, it enables building different controllers through plug-and-play of different applications/systems. SDNator embraces an application- and data-driven design where applications function as data consumers and producers to collectively define the workflows and capabilities of a controller. A data-driven design also allows SDNator to be domain- and protocol-independent, making it possible to integrate with and be deployed in real-world CPS.

Enlightened by previous work on CPS application characteristics and requirements[36], SDNator incorporates two different data backends to provide both event-driven and data-

driven programming patterns. One of them serves as persistent storage to enable historical data usage for offline analysis. To make SDNator more instrumental and efficient, we implement an on-demand data production (§5.2.7) mechanism to give consumer apps control over what data will be produced at what frequency. Fault tolerance and recovery features (§5.2.8) are also built into SDNator for extra reliability. Through extensive benchmarks (§5.4), we show that SDNator (1) delivers over 100K msgs/s using one CPU core on a commodity PC, (2) incurs as little as sub-100us end-to-end delay, and (3) scales with 60 geo-distributed apps and high network latency.

On top of that, we leverage SDNator to implement 3 different control workflows for manufacturing (§5.5.1) and networking (§5.5.2) CPS in our case studies. In particular, in light of the "citizen-supply-chain" phenomenon[125] during the COVID-19 pandemic[1], we demonstrate (§5.5.1.4) how an SDNator-based centralized scheduling workflow can substantially speed up PPE production by 2X without compromising existing production jobs compared to a decentralized uncoordinated approach.

We make the following contributions in this work:

- We design, implement and open-source[2] SDNator, an **extensible, data-driven** framework for enabling centralized control in cyber-physical systems. SDNator achieves extensibility through plug-and-play of apps with no controller adaptations or modifications. SDNator achieves generality through its data-driven abstractions, allowing it to easily integrate with different CPS.
- We design and implement a lightweight client-side library called Data Ubiquity Engine (DUE) that (1) provides **well-defined APIs** to support both **event-driven** and **data-driven** programming patterns, (2) transparently enforces on-demand data production, (3) intelligently performs batching and buffering, and (4) silently monitors application health. DUE enables easy onboarding of heterogeneous apps in SDNator,

[1]Universities, tech firms and 3D print enthusiasts with their own 3D printers respond to the shortage of healthcare workers' personal protective equipment (PPE) by producing it themselves.

[2]https://github.com/Linerd/SDNator.git

delivers over 100K msgs/s and incurs less than 100us delay.

- We demonstrate how easy it is to enable centralized control using SDNator (either through developing new CPS applications or integrating existing ones) in our case studies of manufacturing and networking CPS. In particular, we carry out the first study on digital-twin-equipped centralized control of additive manufacturing fleets and show that it shortens normal production time by up to nearly 40%, and PPE production time by more than 50% compared to a decentralized baseline approach.

## 5.2 Design

In this section, we introduce the design of SDNator. We first establish SDNator's design principles, then give an overview of its architecture and detailed descriptions of its core components. We will also highlight SDNator's on-demand data production and fault tolerance features.

### 5.2.1 Design Overview

Compared to traditional network systems, CPS have some unique characteristics [124, 36] that must be taken into account when designing SDNator:

(C1) CPS have highly heterogeneous machines, protocols, applications, and data.

(C2) CPS are often geo-distributed (*e.g.,* factories, sensor networks) and cross-domain coordination is common.

(C3) CPS applications manifest both event-driven and data-driven patterns.

An overview of SDNator's architecture is shown in Figure 5.1. We make the following key design decisions to accommodate the aforementioned CPS characteristics as well as to address deficiencies of legacy SDN controllers (§5.7):

(D1) Applications are remotely connected to two data backends via Data Ubiquity Engine (DUE, §5.2.6). This allows them to be language-independent (C1), run anywhere

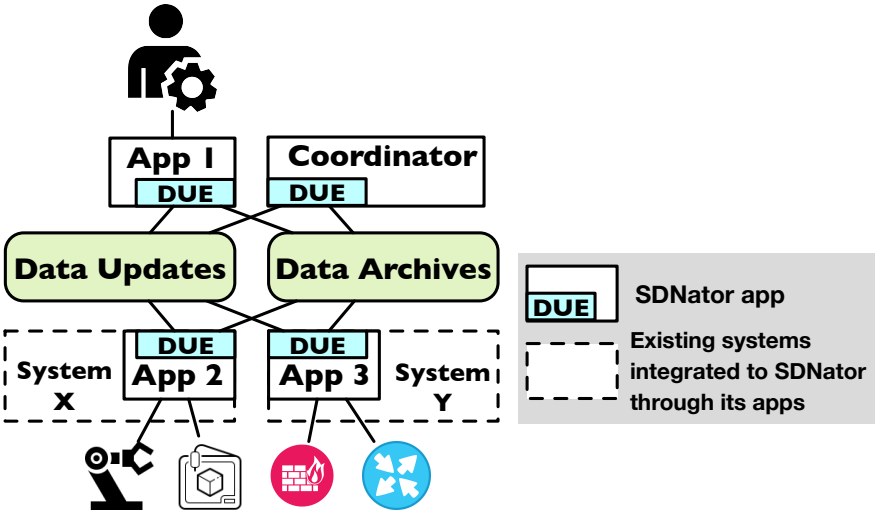



Figure 5.1: SDNator with 3 sample applications (App 1 interfaces with users through northbound APIs; App 2 interfaces with manufacturing devices such as robot arms and 3D printers, App 3 interfaces with networking devices such as firewalls and switches)

rather than bundled with other apps, and scaled independently (C2).

(D2) Apps cooperate as data consumers & producers through a generic (C1) key-value based data schema (§5.2.5).

(D3) Two data backends, Data Updates (§5.2.2.1) and Data Archives (§5.2.2.2), support event-driven and data-driven patterns respectively (C3).

(D4) A special application called Coordinator (§5.2.4) automatically handles the onboarding of new applications, and enables on-demand data production (§5.2.7) based on their interests and capabilities.

### 5.2.2 Data Backends

#### 5.2.2.1 Data Updates

For SDNator applications, Data Updates is the carrier of inter-application communications. It is a high-performance data distribution service that (1) allows applications to publish and subscribe to specific data items, and (2) delivers data items to subscribers in real time. Data Updates itself does *not* store any information, rather, it serves the pur-

pose of notifying interested parties of changes happening in the system (hence the name "Updates") and the details of those changes.

#### 5.2.2.2 Data Archives

On the other hand, Data Archives is a mass persistent storage that stores *all* information that it receives from SDNator applications. It (1) allows applications to store and retrieve any data items, and (2) supports fine-grained and range queries. Data Archives, as its name suggests, serves the purpose of persisting historical data, allowing applications to perform (big-data) analysis and make more informed decisions[36].

### 5.2.3 Applications

SDNator applications, as mentioned earlier, collectively define the specific workflows and capabilities of the "controller". SDNator uses two simple abstractions to categorize different kinds of applications: **producer** and **consumer**, based on whether an application produces data or consumes data (or both). At initialization stage, applications need to register their **interests** (data items that they want to consume) and **capabilities** (data items that they can generate) with the Coordinator (a special application detailed in §5.2.4). After registration, an application can simply call the DUE (§5.2.6) APIs to publish data, query historical data, or subscribe to data of interest. It is worth noting that SDNator does not require applications to be custom-built; rather, existing applications can be easily integrated into SDNator by importing the DUE library. As shown in Figure 5.1, SDNator applications can belong to other existing systems. Through DUE, these applications become the communication endpoints of each system. We demonstrate SDNator's capability to integrate existing systems in our networking case studies in §5.5.2.

### 5.2.4 Coordinator

Coordinator is a special application that handles registration of new applications, monitors application heartbeats (generated by DUE, §5.2.6) and reacts to application failures (§5.2.8). Coordinator sends and receives data via DUE, just like other regular applications. Its interests, quite specially, are the interests, capabilities and heartbeats of other regular applications, and its capability is assignments (specification on what data items a producer should generate and at what frequency, see §5.2.7). Coordinator is only involved during the onboarding process and when apps go offline/unresponsive, therefore it is not on the critical paths of any data production and consumption as they go directly to the data backends. To facilitate users and developers, Coordinator also exposes APIs for querying capabilities and statuses of existing SDNator apps.

### 5.2.5 Data Schema and Specifications

In order for SDNator to accommodate applications from different systems/domains and the highly heterogeneous data they produce, there needs to be a universal and generic data representation. SDNator therefore uses key-value pairs as its data format. In our vision, the data keys must meet the following design goals:

- Uniquely identifies a data item
- Human readable
- Supports wildcarding
- Easily extendable for additional specifications

Figure 5.2 shows an example of SDNator's data key. Inspired by the OpenConfig[82] and gNMI[126] initiatives, which provide a vendor-neutral way of managing network devices and extracting data through a generic, hierarchical key space, SDNator adopts a hierarchical structure for its key format that meets all design goals above. Specifically, a data key is composed of several segments in increasing granularity, followed by optional spec-

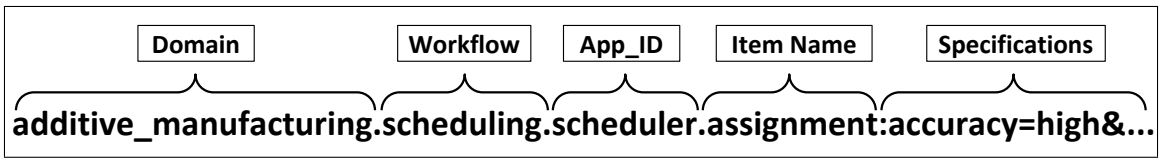



Figure 5.2: Sample data key of production job assignment produced by a scheduler application

ifications. The inclusion of application id allows for different applications producing the same type of data, and makes it easy to search for data produced by the same application using wildcarding.

On top of that, we believe allowing additional specifications on data items can open up new dimensions for application-coordination and controller workflows. For instance, a machine-learning application can specify in its capability the same data item at different accuracies with a tradeoff of time.

### 5.2.6 Data Ubiquity Engine

DUE is the enabler of **event-driven** and **data-driven** programming patterns in SDNator. DUE exposes APIs to SDNator applications to (1) subscribe to a data key and register a callback function with it, (2) publish new values of data keys to Data Updates, (3) persist values of data keys to Data Archives, and (4) retrieve values of data keys specified through queries.

In addition to providing data production and consumption supports, DUE also fulfills the following responsibilities:

- Performs handshake with the coordinator to register an application's capabilities and interests and retrieves its assignments (if a producer)
- Associates subscriber's callback functions with specific data keys and calls them upon Data Updates' notifications
- Paces write requests based on the frequency specified by the coordinator in assignments (§5.2.7)

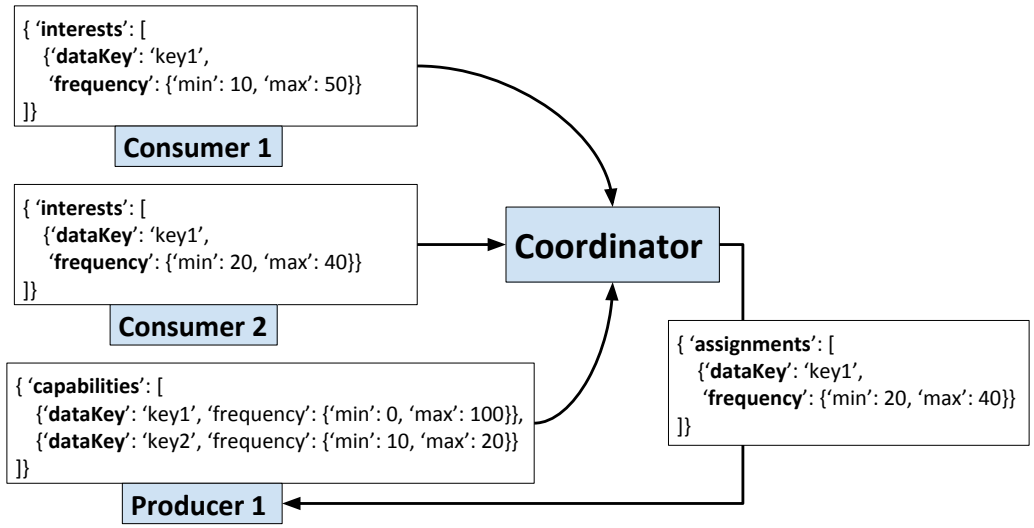



Figure 5.3: A simple example showing the coordinator matching capabilities with interests and generating assignments with adjusted frequencies associated with each data key

- Periodically sends heartbeat signals to the coordinator for application health monitoring (§5.2.8)

We detail our implementation of DUE and how we address several technical challenges to improve application performance in §5.3.

### 5.2.7 On-Demand Data Production

Previous study (ADD[36]) showed that on-demand data production can effectively reduce bandwidth consumption on the southbound channel and latency on data requests in an SDN controller. In SDNator, considering its "flat" architecture, we expand this idea of on-demand data production to all applications.

**Interest consolidation happens automatically in coordinator**. As mentioned earlier, new applications need to register their interests and capabilities with the coordinator during onboarding. Along with the interests and capabilities, applications can also specify *frequencies*, which indicate the rate at which they want to consume or can produce certain data keys. The coordinator, whenever an application joins or leaves, automatically matches all interests and capabilities on a per-data-key basis, and identifies an appropriate

frequency (range) for each data key that satisfies the interested consumers within the producer's capacity. Capabilities that are matched by interests, along with (if any) the adjusted frequencies, are sent back to the corresponding producers as assignments. Figure 5.3 shows a simple example of the above process.

**Frequency enforcement happens transparently and automatically in DUE**. Upon receiving the assignments, DUE automatically parses the them and records the frequency (range) associated with each data key. Whenever a producer calls DUE API to publish a data key, DUE checks whether that data-key is frequency-bound, and paces that write if necessary to meet the frequency specification.

Through on-demand production, SDNator eliminates redundancy in data production (by consolidating all interests) while offering more flexibility and control (See §5.5.2.1).

### 5.2.8 Fault Tolerance, Detection and Recovery

We only consider two sources of faults/failures in SDNator[3]: the data backends and applications. For data backends' reliability, SDNator relies on modern data stores'[127, 128, 129] fault tolerance mechanisms without assuming any better or worse. Here we focus on how SDNator handles failures of applications.

**Fault detection is achieved through periodic heartbeat from DUE to coordinator.** As mentioned earlier, DUE periodically sends heartbeat signals to the coordinator, which allows the coordinator to keep track of the statuses of each application: be it offline, unresponsive, or simply just idle. This process is accomplished, similar to interest and capability registration, by coordinator subscribing to all heartbeat channels of each application. This is feasible based on two key points: (1) Data Updates is publish-subscribe based, therefore sending heartbeat is non-blocking; (2) aggregate heartbeat rate, even with 100

[3]Detecting faults or incorrectness in data is beyond the scope of SDNator's capabilities and should/can be handled by consumer applications.

applications each sending at 10 times per second, will still be orders of magnitude lower than full capacity of an SDNator consumer (see §5.4).

**Fault tolerance is achieved through redundancy in application instances.** In SDNator, the same application can be launched multiple times for redundancy. DUE internally maintains an instance-specific UUID to differentiate between instances. Different instances will register and onboard like any new applications, except that coordinator will associate them with the same application id. Coordinator will randomly select an active instance when generating the first assignments, and sticks to that instance until it goes offline or unresponsive[4].

**Fault tolerance and detection allow for fast fault recovery.** With fault detection, coordinator can quickly perceive when applications go offline or unresponsive. And because redundancy exists, coordinator can easily switch to another active instance in no time to quickly recover from failures. It is worth noting that coordinator itself is not on the critical path of data production or consumption; even if coordinator crashes, it can simply restart and recover its states from new heartbeat information or previous checkpoints in Data Archives, as it only contains *soft state* information collected from DUE.

## 5.3 Implementation and Technical Challenges

In this section, we describe our prototype of SDNator including how we implement Data Ubiquity Engine (§5.3.1) and the frameworks used for Data Updates (§5.3.2) and Data Archives (§5.3.3). We also introduce several optimization techniques we adopt in DUE to address technical challenges that emerge during implementation (§5.3.4).

[4]Different selection strategies can be easily implemented. For instance, pick instances that have the lowest latency or most resources.

### 5.3.1 Data Ubiquity Engine

We implement Data Ubiquity Engine (DUE) as a software library in Python[5] supporting versions 2.7+ and 3.7+, using fewer than 2000 lines of code. DUE can be easily imported in any user applications written in the above environments. The dependencies and development patterns used are readily available and reproducible in other common languages like Java and C++.

The DUE library is composed of three major components: 1) the DUE API that provides applications with generic event-driven and data-driven programming patterns, 2) a Pub-Sub driver that implements the event-driven APIs using Redis Publish-Subscribe functions[130], and 3) a Database driver that implements the data-driven APIs using MongoDB[114].

Being a lightweight client-side library, DUE does not require an adaptation in programming styles like MapReduce as in Hadoop[131] and RDD as in Spark[132], nor does it require complex interactions with job scheduling backends such as YARN[133] (although it can be integrated into SDNator for resource scheduling). Therefore, it is much easier for developers to on-board their applications. Figure 5.4 shows an example of how to import DUE and initialize DUE in an SDNator app with only a few lines of code.

```
from sdnator_due import *
# Optional, configure the Data Archives backend
due.set_db(...)
# Optional, configure the Data Updates backend
due.set_pubsub(...)
# Declare capabilities and interests of the app
caps = [{'dataKey': key1, 'frequency': ['max': 5]}]
ints = [{'dataKey': key2, 'frequency': ['min': 1]}]
# Register app_id, capabilities and interests through DUE
due.init("AppId", PRODUCER | CONSUMER, capability=caps, interest=
    ints)
```

Figure 5.4: Importing and initializing DUE in an SDNator app

[5]for rapid prototyping and decent performance as shown in §5.4

### 5.3.2 Data Updates

For implementing Data Updates, we choose Redis[134] for its high performance and ease of use, as it was used as the message broker backend for some very high throughput and low latency messaging services like Pusher[135]. Specifically, we use Redis' Publish-Subscribe[130] functionality to implement real-time and in-order message delivery, and the event-driven programming interface.

For an SDNator application that wants to publish a new value for a data-key, it can simply call `due.write()` as shown in Figure 5.5.

```
# specify PUB_ONLY to bypass Data Archives
due.write(dataKay, dataValue, [PUB_ONLY])
```

Figure 5.5: SDNator app publishing data through due.write()

Subscribing to a data-key and registering a callback function with it, on the other hand, takes only two lines of code as shown in Figure 5.6.

```
observer = due.observe(dataKey)
observer.subscribe(lambda data: call_back_func(data))
```

Figure 5.6: SDNator app subscribing to data and registering callback function

The `observer` object returned from `due.observe` is an RxPy[136] Subject with powerful data stream capabilities inherited from the ReactiveX programming paradigm[137].

Redis uses TCP for its connections, which has huge implications for SDNator applications: even a modest level of network latency can severely reduce application throughput by limiting the number of communication round trips per second. We address this challenge in §5.3.4.

One potential drawback of using Redis is the lack of delivery guarantee, which is a trade off for its high performance and simplicity. Although we don't observe any losses in our benchmarks (§5.4), if needed, the Data Updates backend can be easily swapped with a reliable message broker like Kafka[129], thanks to DUE's encapsulation.

### 5.3.3 Data Archives

For implementing Data Archives, we choose MongoDB[114] for its relatively good performance and rich feature set for data queries.

When producers call `due.write()`, DUE automatically executes writes to both Data Updates and Data Archives unless otherwise specified (*e.g.,* the `PUB_ONLY` flag shown above). Besides the data key and value, we also include timestamp and application id for the writes to MongoDB, which would facilitate applications such as the coordinator to query data by producer or by time range. Since Data Archives can be slower in write speed, this may cause a slowdown for Data Updates and therefore application throughput. We address this challenge in §5.3.4.

For an SDNator application that wants to retrieve historical data from Data Archives, it can simply issue a `due.get()` call, as shown in Figure 5.7.

```
# fetch all historical records of 'dataKey'
data = due.get(dataKey)
# or use advanced MongoDB queries
data = due.get({
    'dataKey' : dataKey,
    'timestamp': {
        '$gt': someDate
    }
})
```

Figure 5.7: Fetching historical data from Data Archives

Similarly, because DUE's APIs are backend-independent, MongoDB can be swapped with other mass storage databases.

### 5.3.4 Technical Challenges

#### 5.3.4.1 Network Latencies

As mentioned earlier, network latencies can severely reduce application throughput. As described in §5.2.2.1, message ordering needs to be strictly preserved for Data Up-

dates, therefore multi-threading is not an option. Since network latency impacts application throughput, by limiting the number of (TCP) round trips, we can improve throughput by increasing the data volume in each trip (*i.e.,* multiplexing). Since Redis natively supports multiplexing and demultiplexing through its pipelining[138] feature, DUE only needs to process messages in batches and provide internal buffering. To make it more instrumental and flexible, we allow developers to configure their own batch sizes depending on network conditions and communication patterns. Batching can improve application throughput by up to 5X with little latency penalties, as demonstrated in §5.4.2.

#### 5.3.4.2 Slower Data Archives Writes

Data Archives as a mass persistent storage cannot deliver the same write speeds as Data Updates. In order to address this performance gap, we adopt the following two strategies:

**Make Data Archives writes non-blocking.** MongoDB client writes in a blocking fashion. To go around that, we first resolve to Thread Pool. We notice during testing, however, that Python's Global Interpreter Lock (GIL)[139] causes severe performance degradation. Therefore, we switch to Process Pool instead. Though multiprocessing inherently carries the overhead of IPC and data copying which is not ideal, we see in benchmarks (§5.4.2) it delivers satisfying performance. Since this is caused by an inherent limitation of the Python interpreter, it's also safe to assume that our benchmark results serve as a **lowerbound** of DUE's performance.

**Make Data Archives writes more efficient.** Similar to how we optimize Data Updates writes to account for network latencies, we adopt batching and buffering for Data Archives writes as well. §5.4.2 shows results of the improvement achieved through batching.

It is also worth noting that, with Process or Thread Pools, the order of the Data Archives writes won't be guaranteed. To mitigate this, DUE silently appends a timestamp to each data item being written to Data Archives so that order can be restored during future queries. And to make queries faster, we have adopted common database optimization techniques

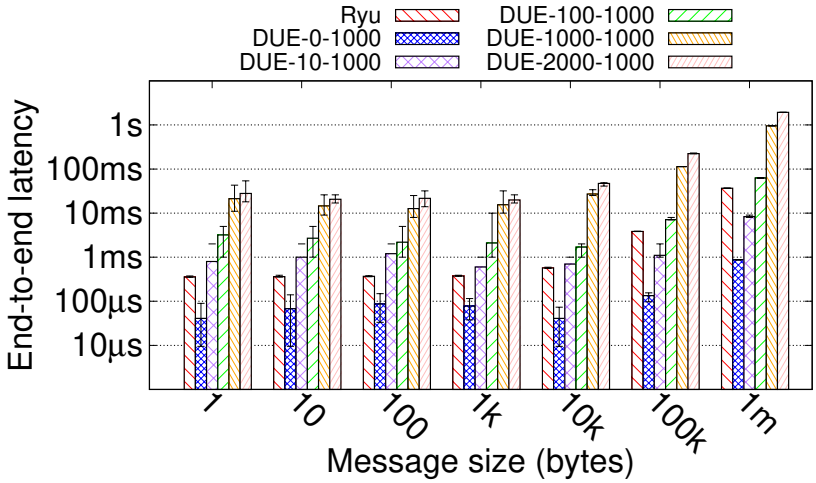



Figure 5.8: A comparison between Ryu and SDNator on end-to-end latency between a producer and a consumer when sending messages at different sizes. For SDNator: DUE-{Data Updates batch}-{Data Archives batch}.

like indexing[140]. Applications are allowed to attach a `index: true` to their capabilities (§5.2.4), so any `due.get()` requests toward those items can be performed much faster.

## 5.4 Benchmarks

In this section, we demonstrate SDNator's performance (Figures 5.8, 5.9, 5.10) and scalability (Figures 5.12, 5.13) through a series of benchmarks. These benchmark results help quantitatively evaluate the latency overhead, achievable application throughput, and scalability of the system. Unless otherwise specified, benchmarks are run on an Ubuntu 18.04 (Linux 4.15.0) machine with Intel Core i7-7700K@4.2GHz quad-core processor and 32GB of RAM. Each experiment is repeated ten times.

### 5.4.1 End-To-End Latency

Unlike existing SDN controllers[29, 30, 33, 47] where applications (especially built-in services like *topology discovery* and *routing*) typically communicate via inter-procedure calls, SDNator uses remote-procedure calls and introduces a path stretch through the Data

Updates (§5.2.2.1). In order to understand the latency "penalty" potentially imposed by SDNator's design, we measure the inherent (when communicating through the loopback interface of the host machine) end-to-end latency (*i.e.,* duration between a batch of messages is sent and received) between two SDNator applications when sending messages of different sizes using different batching strategies. We randomize the bytes in each message to eliminate potential caching influences. For comparison, we also include results from Ryu[29], a widely used SDN controller written in Python. We write two applications in Ryu and leverage Ryu's own event APIs to publish and subscribe to our custom events that contain those messages of various sizes. The results are shown in Figure 5.8.

From the figure we can see that SDNator's base latency (when no batching is enabled for Data Updates) is even lower than Ryu's. Even at a batch size of 100, SDNator is still comparable to Ryu.

**Finding 1.** SDNator has a lower inherent latency footprint than Ryu. Even with moderate batching enabled, SDNator can still deliver comparable performance.

It is worth noting that we have introduced a `NO_WAIT` keyword to `due.write()` and a flushing API such that writes can be executed right away even when batching is enabled to further reduce unnecessary latency overheads.

### 5.4.2 Application Throughput

Achieving high throughput between applications is both critical and challenging for SDNator. As described in §5.3.4, Data Ubiquity Engine (DUE) adopts techniques like batching and buffering to improve application throughput. How effective are these techniques? How much additional overhead does DUE incur? To answer these questions, we conduct several experiments (same setup as above, measuring throughput instead of latency) with our prototype using different DUE configurations detailed below:

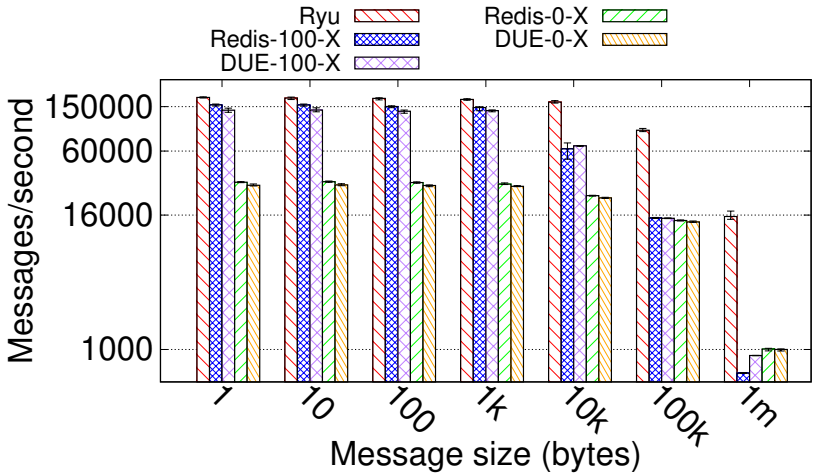



Figure 5.9: Application throughput when using Ryu, raw Redis APIs w/ and w/o batching, and DUE w/ and w/o batching (Data Archives disabled)

#### 5.4.2.1 Data Updates Batching

In the first experiment, we focus on the effectiveness of Data Updates batching and the overhead of DUE. We measure and compare application throughput (in messages per second) when using Ryu, raw Redis pub-sub APIs with and without batching, and DUE with and without batching. Based on results in Figure 5.8, batch size is set to 100 for both Redis and DUE. Results are shown in Figure 5.9.

We can clearly see that DUE+Redis performs fairly close to raw Redis at batch size 100. In fact, the difference is less than 10%. We can also tell that batching has up to 5X improvement on throughput. Ryu does deliver a higher throughput than Redis and Redis+DUE, but the difference is marginal until the message size climbs up to more than 10KB when Redis is bound by bandwidth of the loopback interface while Ryu is mostly bound by memory speed. In Ryu, events typically carry packets that are less than 1.5KB, therefore SDNator is fairly close.

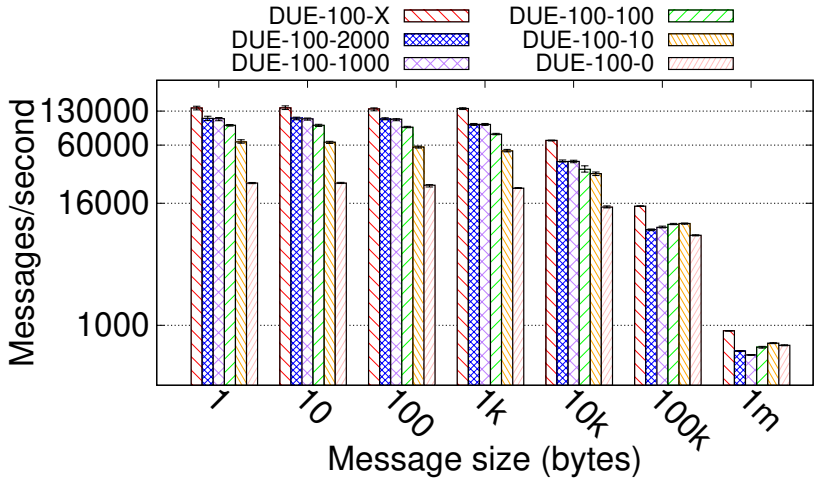



Figure 5.10: Application throughput w/ different Data Archives batch sizes (Data Updates batch size set to 100)

**Finding 2.** DUE incurs 10% or less overhead compared to raw Redis APIs.

**Finding 3.** A Data Updates batch size of 100 can improve app throughput by up to 5X in both DUE & Redis.

**Finding 4.** Ryu delivers higher application throughput than SDNator, although the gap only becomes substantial when message sizes are 10KB+.

#### 5.4.2.2 Data Archives Batching

In the second experiment, we focus on the overhead of Data Archives (given it is slower than Data Updates) and the effectiveness of Data Archives batching. We measure and compare the application throughput with different Data Archives batch sizes, as shown in Figure 5.10.

**Finding 5.** Enabling Data Archives slows down application throughput by ~20% to ~80% depending on the Data Archives batch size.

**Finding 6.** A Data Archives batch size of 1000 can improve application throughput by up to 3.4X.

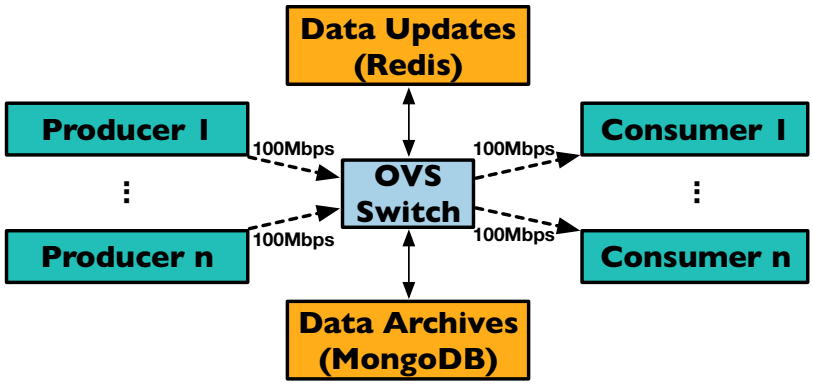



Figure 5.11: Mininet emulated topology (described in §5.4.3); links marked with dotted arrows have 100Mbps bandwidth and latency from 0 to 50ms

### 5.4.3 Scalability

To further test SDNator in a more realistic setting with network latencies and evaluate its scalability, we carry out another experiment in which we use Mininet[85] to emulate a star topology with applications running on different hosts with 100Mbps bandwidth, as shown in Figure 5.11. This experiment is run on an Ubuntu 18.04 (Linux 4.15.0) server with Intel Xeon E5-4620v2@2.60GHz 8-core processor and 128GB of RAM. The message size is fixed at 1000 bytes and Data Updates batch size 12500. The number of pairs of producers and consumers is from 1 to 30, and the one-way latency between producer/consumers and the switch varies from 0 to 50ms (*i.e.,* end-to-end latency from 0 to 100ms). We measure individual application throughput on each host to see if they are fairly close, and aggregate them to see if overall performance suffers as contention increases. Results are shown in Figures 5.12 and 5.13 for individual and aggregate throughput respectively.

If we look at the results for up to 15 producer/consumers in Figure 5.12, when individual throughput is only capped by bandwidth, we can clearly see that (1) higher lantencies incur moderate drop on throughput, and (2) more applications does not lead to either lower throughput or high fluctuations in throughput. As we move to Figure 5.13, we can further tell that aggregate throughput does not drop even with 40 applications. Even with 60 applications, we only see a slight drop in aggregate throughput likely due to limited computing

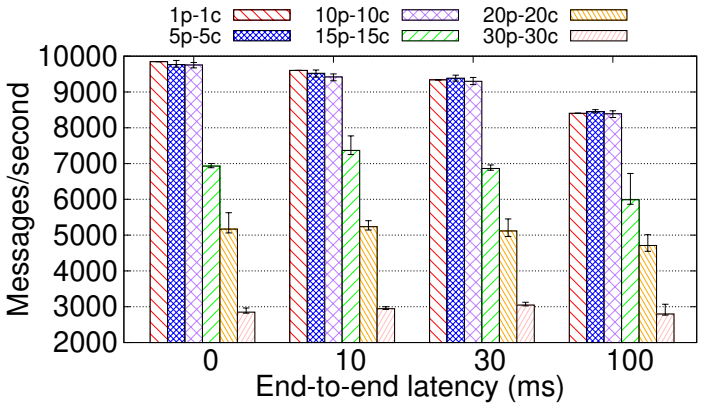



Figure 5.12: Individual application throughput under different network latencies and # of consumer/producer pairs (Data Archives batch size set to 12500)

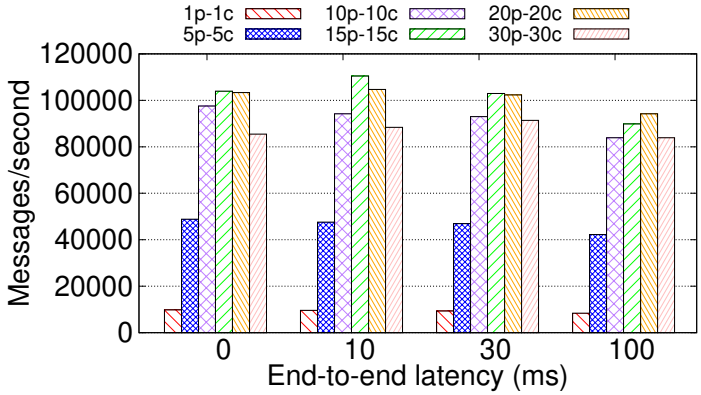



Figure 5.13: Aggregate application throughput under different network latencies and # of consumer/producer pairs (Data Archives batch size set to 12500)

resources of our machine.

Thanks to SDNator's modular design and the incorporation of state-of-the-art data stores, its scalability can be further improved through replication[128] and clustering[127] which are common practices today. For example, operators managing geo-distributed sites can opt for running local instances of data backends to reduce the impact of network latencies and let the backends' clustering and replication mechanisms take care of the synchronization.

**Finding 7.** Higher network latencies lead to slightly lower application throughput; however, even with 100ms end-to-end latency, individual applications can still achieve ~67% utilization of 100Mbps bandwidth.

**Finding 8.** Contention does not cause inequalities or degradation among SDNator applications

**Finding 9.** SDNator scales well to a fairly large number of applications without noticeable degradation.

## 5.5 Case Studies

### 5.5.1 Additive Manufacturing

As mentioned earlier (§5.7), the idea of centralized control has been explored in the context of smart manufacturing which relies heavily on cyber-physical systems (CPS) for their reconfigurability and data availability. Currently, no existing centralized control framework for CPS is available to allow researchers or developers to easily test out their applications. In this case study, we focus on one category of smart manufacturing: additive manufacturing, *a.k.a.* **3D printing**. More specifically, we perform the first-ever study of digital-twin-based control of an additive manufacturing fleet [27] by leveraging SDNator to build centralized control workflows for multiple 3D printing systems. Our study clearly suggests that a digital-twin-equipped centralized controller helps reduce production time (§5.5.1.2) as well as detect and react to anomalies in real time (§5.5.1.3). Moreover, we evaluate our SDNator-based controller in a more complex scenario where urgent requests such PPE[125] or ventilators[141] are added on top of existing production jobs and demonstrate that it adjusts production plans on the fly to speed up PPE production by 2X without compromising existing production jobs (§5.5.1.4).

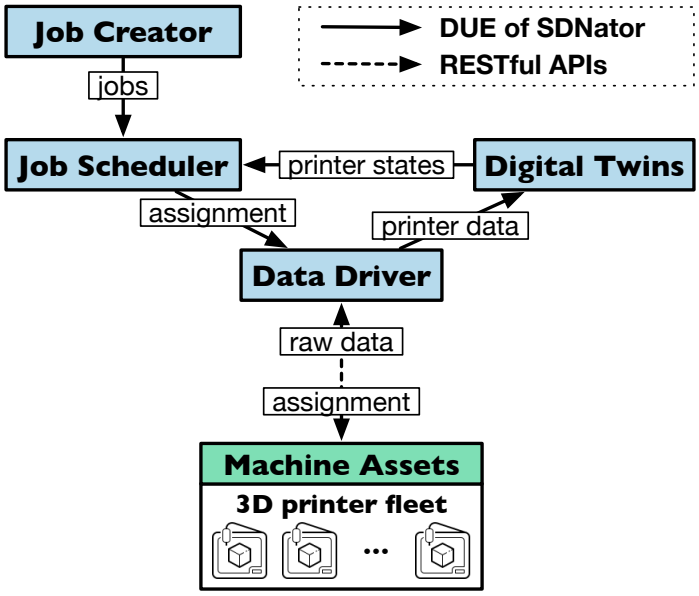



Figure 5.14: An SDNator-based centralized control workflow for additive manufacturing scheduling: Each blue box is an SDNator app. The Job Scheduler subscribes to different amount of information from Digital Twins to implement different scheduling algorithms.

#### 5.5.1.1 An SDNator-based Controller for 3D Printing

Figure 5.14 shows the workflow of the central controller built using SDNator, which consists primarily of the following SDNator apps:

**Job Creator** initializes the production orders (both regular jobs and PPE jobs) at predefined times.

**Job Scheduler** receives the production orders from the Job Creator and the state information of the fleet from the Digital Twins (DT) app and executes a scheduling algorithm (described below) to assign jobs to machines in the fleet.

**Digital Twins.** A machine-state DT monitors the functional state (*e.g.,* idle, running, *etc.*), physical properties, and time information of a printer. A queue DT monitors the job queue and updates the queue finish time, queue length, average queue wait, and queue history. An anomaly detection DT monitors the sensors on the printer to detect anomalies, and reports any anomalies to a fleet DT. When there is an anomaly, the fleet DT schedules a

maintenance event, which stops the current job to save time and material and the current job is rescheduled according to the scheduling algorithm. The DT app monitors each printer, sensor, and queue to update machine-states, queue, and anomaly detection DTs.

**Data Driver** sends the job assignments to the AM fleet through a **Machine Assets** app that faithfully emulates the Jedi RESTful API of Ultimaker 3[142] Fused Deposition Modeling (FDM) printers, which are off-the-shelf printers commonly used in practice. The Data Driver also transfers data from the Machine Assets to the DT app so that DTs maintain the up-to-date state of the printers. The Machine Assets manages an emulated fleet of 3D printers modeled[143] based on the Ultimaker 3 FDM printer. Each printer has a job queue and two sensors for anomaly detection.

To demonstrate the performance benefits of centralized control for 3D Printing, we implement 4 scheduling algorithms that leverage the central controller to different extents:

**Decentralized** does not utilize any global view provided by SDNator. Each machine takes a production order when queue is empty with no knowledge of each other. This is also the most common PPE production scheme during COVID-19 where contributors with existing AM capabilities produce PPEs for healthcare workers in an uncoordinated fashion.

**Centralized-Baseline.** The scheduler is aware of all the machines in the AM fleet thanks to SDNator making the fleet size captured by Data Driver available to other apps. It is however not using the states of the machines that Digital Twins are publishing. As a result, the Job Scheduler simply distributes the production orders to all the machines in the fleet uniformly.

**Centralized-FCFS.** The Digital Twins publish realtime state information (queue, machine-state, and anomaly of each machine) for other applications to consume. The scheduler uses the availability and queue length of the printers to schedule each job in a production order to a machine in the fleet, in a first come, first served (FCFS) fashion.

**Centralized-Dynamic** minimizes the expected makespan of the jobs through optimization by making use of all the machine-state information published by the Digital Twins. Specifically, we form an integer linear program (ILP) to evaluate the optimal number of jobs to be sent to each machine based on their availability, setup time, and expected queue finish time.

#### 5.5.1.2 Shortening the Production Time

To begin, we consider a baseline production scenario where the AM fleet executes normal orders without anomalies or demand changes. Figure 5.15 shows the performance of each scheduling algorithm, in terms of makespans. Final results are normalized by the maximum average makespan of the worst-performing algorithm for ease of comparison. The decentralized algorithm performs the worst as expected since jobs are assigned to machines randomly. The centralized baseline solution outperforms the decentralized one and performs comparably to the centralized FCFS solution. The centralized dynamic solution excels as it optimizes job assignments to minimize expected makespan, potentially saving hours, days or even weeks.

**Finding 10.** The benefit of a global-view provided by the SDNator-based central controller is evident from the baseline results. The schedulers that utilize a centralized approach can greatly improve the makespans in the baseline scenario.

#### 5.5.1.3 React To Real-Time Anomalies

Here we consider a production scenario with probabilistic random anomalies on printers while producing the same production orders as in the baseline scenario (§5.5.1.2). If an anomaly is detected during a print, the same job needs to be re-printed. Figure 5.16 shows the performance of each strategy, in terms of makespans. Similar to §5.5.1.2, the benefit of a global view is pronounced by the shorter makespans of the centralized solutions. When

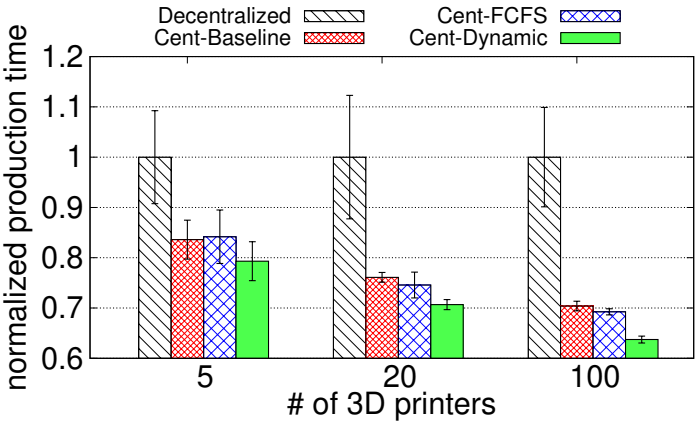



Figure 5.15: Comparison of normal job makespans between decentralized and centralized scheduling algorithms

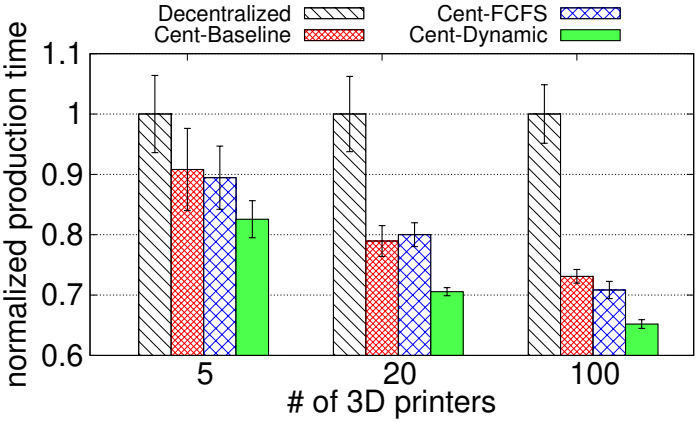



Figure 5.16: Comparison of job makespans between decentralized and centralized scheduling algorithms with anomalies

an anomaly is detected by the anomaly detection DT, the current job is reprinted on the same printer for the decentralized scheduler, and scheduled to (potentially) another printer based on the strategy of the centralized schedulers.

As the number of machines in the fleet increases, the benefit of the global view through the DTs becomes more evident. The run-time information provided by the DTs improves the efficiency of the dynamic scheduler at all scales.

**Finding 11.** SDNator-based central controller improves reliability by enabling digital-twins that capture real-time information such as machine status and occupancy for quick reaction to anomalies and effective mitigation.

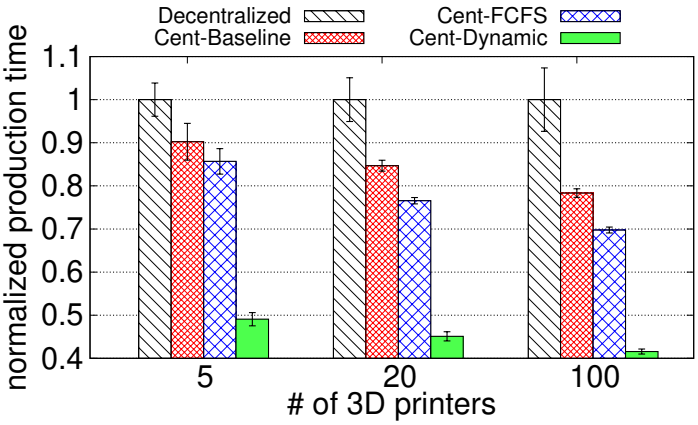



Figure 5.17: Comparison of PPE makespans (w/ normal jobs) between decentralized and centralized scheduling algorithms

#### 5.5.1.4 Adjust Production Plan On The Fly

Finally we consider a production scenario where an urgent request of Personal Protective Equipment (PPE) is received while the fleet is maintaining baseline production described in §5.5.1.2. Figure 5.18 shows the final makespan of each scheduler and Figure 5.17 shows the makespan of the PPE orders.

The PPE production scenario is of high importance and requires timeliness due to its wide applicability during the COVID-19 pandemic. As PPE production has high priority, it is expected that a dynamic scheduler should be able to prioritize these orders over the preexisting orders in the AM fleet. While all the other schedulers utilize a basic FIFO queue, the dynamic scheduler utilizes a reconfigurable queue where existing jobs in the queue can be preceded by priority jobs. The dynamic scheduler uses the same ILP-based makespan minimization optimization to schedule the PPEs with priority while balancing the total production makespan. The results in Figure 5.17 show that this approach enables the dynamic scheduler to produce the PPEs ~50%-100% faster than the other schedulers, without sacrificing overall makespans as shown in Figure 5.18.

**Finding 12.** SDNator-based central controller provides more agility to adapt to changes in production demands while preserving efficiency.

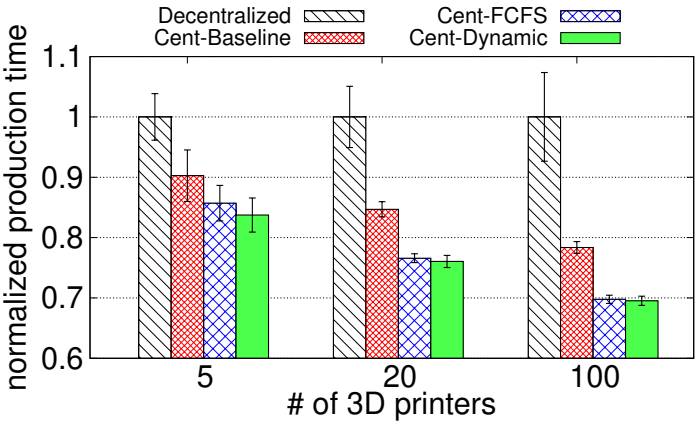



Figure 5.18: Comparison of final job makespans (w/ PPE) between decentralized and centralized scheduling algorithms

### 5.5.2 Networking

To further demonstrate SDNator's generality and extensibility, we implement two control workflows in networking systems by converting existing networking applications into data producer/consumers and plug-and-play, as shown in Figure 5.19.

#### 5.5.2.1 Network Telemetry and Monitoring

Gupta *et al.* [19] show via their Sonata framework the power and flexibility of using SDN techniques like P4[17] and big data systems like Spark[132] to perform network telemetry at scale. It occurs to us that Sonata is an ideal upstream SDNator producer to efficiently generate and stream digested information to downstream consumers. More specifically, we can interface Sonata's custom querying capability with SDNator's on-demand data production mechanism (§5.2.7) to allow other applications to determine what information to extract from network traffic at a fine granularity by specifying interests that can be mapped into specific queries in Sonata, *e.g.,* `network.traffic_capture.sonata.tcp:flag=syn` using SDNator's data schema (§5.2.5).

By simply importing DUE into Sonata, we manage to export from Sonata P4-processed packets or datagrams filtered by Spark queries. In addition, since Sonata directly interfaces

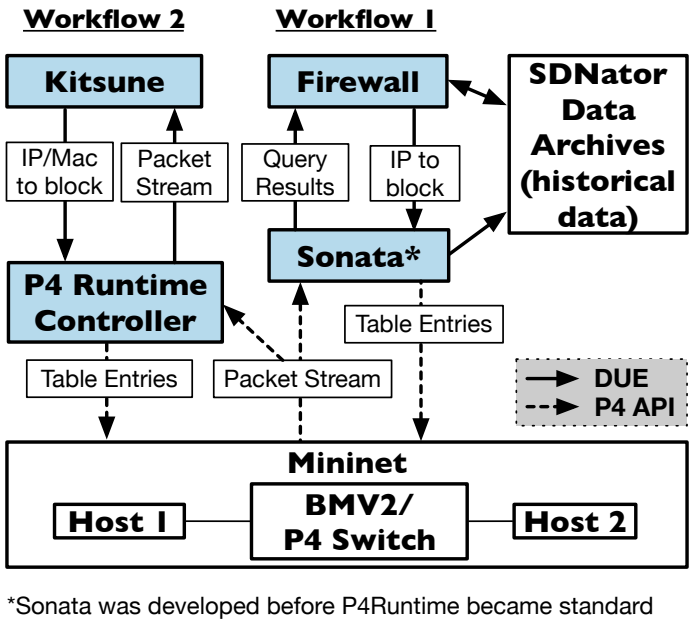



Figure 5.19: Control workflows constructed by using SDNator to integrate different existing networking systems

with P4 switches, we are also able to convert Sonata itself into an actuator of external P4 commands[144] that can be deployed on the switches it's monitoring. In companion to that, we develop a proof-of-concept firewall application that (1) consumes info of suspicious IP addresses (*e.g.,* SYN Flood attack[145]) from Sonata queries, (2) matches it against **historical records** stored in SDNator's **Data Archives**, and (3) sends reconfiguration commands to Sonata to block attack traffic. See Workflow 1 in Figure 5.19.

Albeit an extremely simplified showcase of consuming both historical data and real-time data, the above use case can be profoundly extended by incorporating formal flow-based (offline using SDNator's Data Archives) or event-based (online using SDNator's Data Updates) methodologies described by Mooer *et al.* [146].

#### 5.5.2.2 Real-time Intrusion Detection and Mitigation

Intrusion Detection Systems (IDS) have been popular targets[147] of the network security community and more recently of SDN[148]. For the second use case, we expand on an online, machine-learning-based network intrusion detection system (NIDS) called

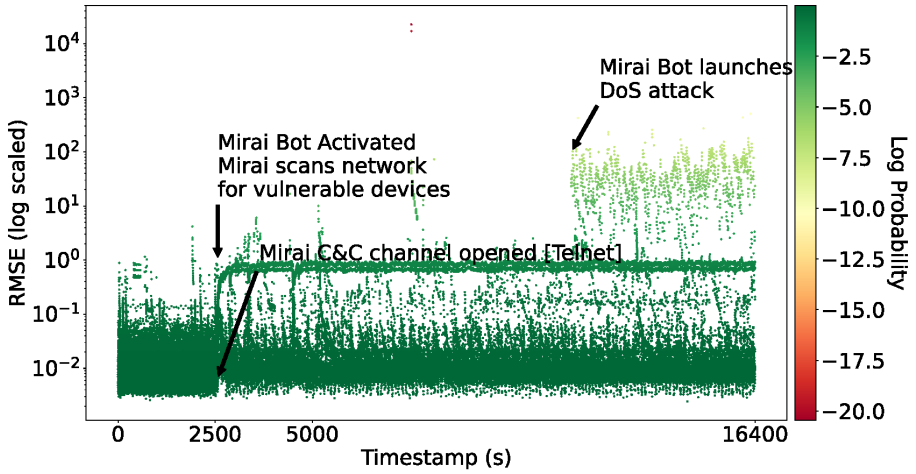



Figure 5.20: Using Kitsune as a standalone intrusion detection tool for Mirai botnet attacks

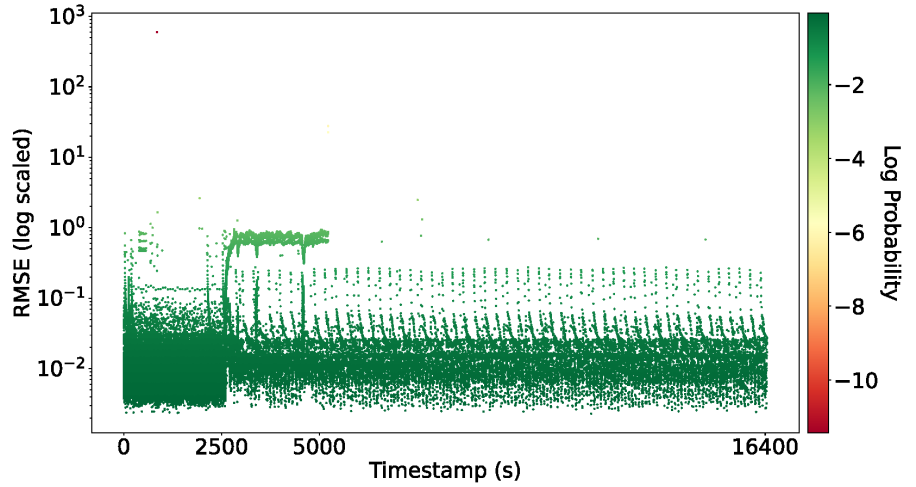



Figure 5.21: Using Kitsune in conjunction with P4Runtime to detect and block botnet attack traffic, enabled by SDNator

Kitsune[149] and implement a real-time intrusion detection and mitigation workflow (see Workflow 2 in Figure 5.19). We use P4 BMV2[150] software switches to emulate the data plane, and P4Runtime[32] as the controller for the P4 switches. We stream packets from P4 switches to the P4Runtime controller and then pass over the packet stream to Kitsune using DUE; in return the controller receives analysis results from Kitsune via DUE and writes the results (either IP or MAC addresses to block) as flow table entries onto P4 switches.

To verify the effectiveness of the implemented workflow, we test it with a simulated Mirai[151] botnet DoS attack using a dataset from Kitsune in an emulated network. By

default, Kitsune uses the first ˜55k of packets in the stream to train its ML model, after which it will give an RMSE (anomaly) score to each packet. In Kitsune, an RMSE score of 1 or above can be considered an anomaly, so we modify Kitsune to publish source IP or MAC addresses of those packets scoring over 1. Subscribing to those results, P4Runtime instructs the P4 switches to drop packets from those addresses. Figures 5.20 and 5.21 show time-series plots of the scores when Kitsune runs alone and as part of our workflow respectively. We can clearly see the difference made by the mitigation module from both figures. It is worth pointing out that we updated **fewer than 10 lines of code** in Kitsune to include it in our workflow using SDNator.

#### 5.5.2.3 Takeaways

There are a couple key observations we can make based on our networking case studies:

- Application integration with SDNator is usually straightforward with fewer than 10 lines of Python code.
- SDNator can easily interconnect heterogeneous modules/frameworks (*e.g.,* a P4 SDN controller and an IDS system) through its generic data-driven pipelines and compose them into cooperative workflows.
- A common one-to-many data flow, where multiple apps consume the same packet stream, is naturally and efficiently supported by SDNator's publish-subscribe model.
- SDNator readily facilitates both online and offline data analysis use cases with Data Updates and Data Archives
- On-demand data production opens up unique dimensions for consumers to customize upstream data producers.
- Data-intensive workflows like real-time IDS can be supported with satisfying performance.

## 5.6 Discussions

### 5.6.1 Security and Privacy Concerns

In SDNator, all applications communicate through DUE and the data backends. Without access control, an application can send data to and receive data from arbitrary applications, which could be exploited by malicious or compromised applications. Although security and privacy are not our main focus of this work, and SDNator shares the same assumptions with existing SDN controllers that applications are run by trusted parties, we suggest a few changes that could increase the security of the system:

- The Coordinator maintains a per-key whitelist of allowed apps to restrict access to data keys.
- Set up a public authentication server that authorizes and manages access tokens for new apps.
- Add a proxy between data backends and DUE to enforce access control to the data backends by checking the access tokens as well as Coordinator's whitelist.

### 5.6.2 Limitations

As a distributed system, SDNator application throughput will be capped by available bandwidth. Since Data Updates and Archives can be scaled out, one could instantiate local data backends to facilitate applications in close proximity.

SDNator is also not an ideal tool for session-based communications where data flow is strictly point-to-point such as between clients and servers. Frameworks like gRPC[152] are more suited for these scenarios.

Even though SDNator embraces data-driven use cases, it is not intended to be used to transfer data for the sake of transferring data, especially in bulk volumes. Traditional client-server solutions are more reliable with break-point resumes and caching.

## 5.7 Related Work

**Legacy SDN controllers.** A variety of SDN controllers, such as Ryu [29], Floodlight [30], OpenDaylight [33] and ONOS [47, 34], are available in different programming languages. Legacy SDN controllers (Figure 1.1) are designed as monolithic ecosystems where internal applications are mostly written in the same programming language and run alongside each other (*i.e.,* scaling is achieved through multiplying controller instances rather than apps). Protocols (*e.g.,* OpenFlow [1]), device abstractions (*e.g.,* switches), and data types (*e.g.,* packets) are predominantly tailored to networking applications. Moreover, applications produce and consume data (which typically come in the form of events) in a publish-subscribe fashion.

**SDN-like control architectures in CPS.** Previous works have explored SDN-like centralized control in many different CPS such as smart manufacturing[26], IoT[23], autonomous automobile[25, 24], and storage[153]. These works focus on addressing specific challenges in each domain in a centralized fashion, and rely on either simulations or proprietary prototypes for proof-of-concept. SDNator is an open-source framework that enables researchers and developers to easily build such centralized controllers for CPS.

**Data-driven controller design.** In ADD [36], Lin *et al.* present a case study of manufacturing systems and applications and identify the deficiencies of legacy SDN controller designs in handling data-driven use cases. They propose a data-driven controller design that aims to address those deficiencies, with limited implementation and evaluation. SDNator does share ADD's vision regarding the importance of supporting data-driven use cases in a controller, but SDNator has a drastically different architecture compared to ADD and SDNator is fully implemented, well tested and open-source.

**Hierarchical and distributed controller designs.** Previous studies such as SoftMoW [62], ElastiCon [154], and WE-Bridge [155] improve SDN controller scalability by arranging multiple controller instances in hierarchical or distributed manners. SD-CPS [124] uses message-oriented-middleware (MOM) to enable inter-domain communi-

cation between multiple CPS controller instances. Besides not addressing data-driven use cases like SDNator does, these works also differ from SDNator in their architecture: SDNator is not a monolithic controller or a cluster of such controllers; it is a cluster of applications loosely connected by its data backends, therefore the applications can scale independently.

## 5.8 Summary

In this work, we go beyond building yet another "SDN controller". Instead, we provide researchers and developers an extensible, data-driven, scalable and easy-to-use platform to implement/integrate their applications and "create" their own centralized controllers for a cyber-physical system (CPS). SDNator supports both event-driven and data-driven programming patterns and allows apps to leverage both real-time data streams and historical data. Benchmarks show that SDNator delivers comparable performance to Ryu with minor overhead. We demonstrate SDNator's usability and generality through our case studies on networking and manufacturing CPS. Using SDNator, we carry out the first study on dititaltwin-based control of additive manufacturing fleets and see substantial performance gains in shortening job makespans in various scenarios.

To our knowledge, SDNator is the first open-source framework for CPS to enable easy development & integration of apps for device management and data analysis in a centralized fashion. As the boundary between physical and traditional networking systems continues to blur (*e.g.,* the increasing adoption of IoT devices), a platform that is domain-agnostic, highly scalable and easy to use will become even more relevant.

# CHAPTER VI

# Conclusion & Future Work

This chapter concludes the dissertation. We highlight the key contributions of this dissertation, discuss limitations of the proposed solutions, and describe potential future research directions based on the works presented.

## 6.1 Key Contributions

In this dissertation, we describe the design and implementation of systematic solutions to enable and improve centralized control in network and cyber-physical systems without tailoring to specific applications. We demonstrate the effectiveness of these solutions by both incorporating existing applications and developing new ones. Specifically, we:

- **Expand SDN programmability to support in-network buffering.** We design Programmable Buffer as a data-plane component and a set of well-defined APIs to enable applications to control where, when, and how to pause and resume a network flow. By implementing PBs as virtual network functions attached to programmable switches, we manage to preserve full compatibility with existing SDN applications while supporting the new functionalities. Using PB, we are able to develop a new mobility management application and a connection-less communication service for emerging 5G use cases.

- **Generalize SDN traffic management abstractions.** We study over 200 MOPs from a major U.S. carrier to understand the commonalities and differences between different network functions in terms of traffic migration. We are able to identify a few key parameters and steps that suffice to describe all traffic migration procedures. Based on these findings, we design Egret to provide generic interfaces and modular workflows for both network operators and vendors. Egret not only allows us to perform traffic migration through a simple, unified abstraction, it also facilitates reverse traffic migration (*i.e.,* rollback) and job parallelization.

- **Develop an extensible framework for building centralized controllers in networks and CPS.** We systematically compare traditional network systems with smart manufacturing systems to identify their key differences in application behaviors. We find that CPS applications are predominantly data-driven, as will future network applications. We design SDNator to support data-driven applications in general and to integrate existing applications and onboard new applications easily. We leverage SDNator to carry out the first study of digital-twin-based centralized control of additive manufacturing fleets and show that centralized approaches significantly outperform distributed approaches in production speed and flexibility. In particular, SDNator allows us to realistically explore centralized solutions for optimizing the production schedule to address the PPE shortage problem during the COVID-19 pandemic.

## 6.2 Limitations and Mitigations

The work of this dissertation is based on the premise that an SDN-like centralized control paradigm can improve the performance (thanks to the global visibility) and agility (thanks to the software-driven model) of network and cyber-physical systems compared to traditional distributed approaches. There are, however, substantial trade-offs between centralized and distributed control solutions that should not be neglected. Here we emphasize

two inherent limitations of centralized approaches and discuss how we mitigate them in our proposed solutions.

### 6.2.1 Increased Control Latency

By consolidating individual devices' control-plane into an *external* controller, centralized control approaches inadvertently increase the control latency for each device compared to distributed approaches. A low control latency is critical for latency-sensitive applications such as mobility management in cellular networks (as introduced in Chapter II and anomaly detection in various safety-critical cyber-physical systems.

**Programmable Buffer.** As shown in §2.6.2, PB's control latency meets 5G's strict requirements. In addition, PB allows further decoupling between the control-plane and data-plane, meaning latency-sensitive actions that don't require global visibility can be offloaded to the data-plane and performed by PB locally (*e.g.,* inter-buffer synchronization for fast mobility management as described in §2.6.3.2).

**Egret.** Traffic migration, especially in the context of change management, is not a latency-sensitive operation because traffic convergence among other processes are orders of magnitude slower than the control latency. Moreover, Egret achieves significant time reduction with its automated workflows, and can detect and react to network dynamics swiftly thanks to its synergy with the SDN control-plane (*e.g.,* switch failure mitigation as shown in §3.4.2).

**SDNator.** SDNator applications are fully decoupled, which means they can run in close proximity to physical devices for low control latency regardless of the location of other applications or data backends. They can even run in a distributed fashion if needed. This design allows SDNator applications to strike a balance between global visibility, coordination, and low control latency.

### 6.2.2 Single Point of Failure and Bottleneck

Another drawback of consolidating control is that a central controller becomes a single point of failure. If not used properly, it can also be a bottleneck (*e.g.,* using controller to buffer network packets as shown in §2.6.3). Hierarchical controller designs (*e.g.,* SoftMoW [62]) and controller replication (*e.g.,* ONOS [47]) techniques can effectively mitigate these two limitations, which also apply to work of this dissertation. In addition, SDNator allows replication at application-level and adopts several failure detection and recovery mechanisms which can also improve the reliability and performance of the centralized control-plane.

## 6.3 Future Work

This dissertation lays down the system foundation for innovations on the application side. While we only presented a few examples in this dissertation, we look forward to many exciting and "out-of-the-box" explorations in the emerging network systems and cyber-physical systems. Here we highlight a few possible directions:

- **Using Programmable Buffers for more than buffering.** In Chapter II, we describe how applications can use PB to pause and resume network traffic. We implement PB as virtual network functions so that we can leverage the existing SDN traffic redirection capabilities. More importantly, these PB "boxes" have full access to each packet, which means we can implement additional functionalities such as packet inspection and traffic monitoring in them while keeping the programming abstractions intact. Further decoupling the control-plane and data-plane allows more latency-sensitive operations to be executed on the data-plane.

- **Optimized traffic migration job scheduling using Egret.** In Chapter III, we introduce Egret's mask-based representation that automatically keeps track of job states

and enables job interleaving. Although we anticipate time reduction as a result of parallelization, the actual impact and optimal strategy (including job conflict detection) for job scheduling are operation-, topology- and policy-dependent. We encourage explorations on this route.

- **Centralized control in CPS.** There are many similarities between traditional network systems and various CPS that make the transition to a centralized control paradigm in CPS plausible. We use manufacturing systems as an example in Chapter IV, but it also applies to other CPS. For example, autonomous/connected vehicles are like "packets" that travel on roads rather than links, and intersections are like routers or switches. Many techniques from the network domain, such as routing and traffic engineering, can potentially help mitigate traffic congestion, reduce travel time, and prevent traffic accidents in the real world. While distributed approaches excel in making localized decisions fast, centralized approaches are more flexible, agile, and optimized. With continuous advances in network connectivity technologies (*e.g.,* 5G), the latency penalty of a centralized approach can be further reduced.

- **Adopting digital twins in control workflows.** DT technology is one of the key enablers of Industry 4.0 and smart manufacturing. We show in Chapter V a concrete additive manufacturing example using SDNator with simple DT applications to facilitate analysis and decision making. In production, the number of different DTs and their complexity are much higher, which will introduce immense challenges in constructing workflows consisting of many different DTs that produce and consume different types of data. It is also non-trivial for a decision-maker-type application to consolidate information from different other applications, including the DTs, especially when conflicts are present.

# BIBLIOGRAPHY

# BIBLIOGRAPHY


[1] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[2] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 183–197, New York, NY, USA, 2015. Association for Computing Machinery.

[3] Arjun Roy, Deepak Bansal, David Brumley, Harish Kumar Chandrappa, Parag Sharma, Rishabh Tewari, Behnaz Arzani, and Alex C. Snoeren. Cloud datacenter sdn monitoring: Experiences and challenges. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, page 464–470, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.

[5] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery.

[6] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special*

*Interest Group on Data Communication*, SIGCOMM '18, page 74–87, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Onap. `https://www.onap.org/`. (Accessed on 08/09/2020).

[8] L. Peterson, A. Al-Shabibi, T. Anshutz, S. Baker, A. Bavier, S. Das, J. Hart, G. Palukar, and W. Snow. Central office re-architected as a data center. *IEEE Communications Magazine*, 54(10):96–101, 2016.

[9] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[10] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 19, USA, 2010. USENIX Association.

[11] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 365–378, USA, 2010. USENIX Association.

[12] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, page 29–42, USA, 2013. USENIX Association.

[13] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. Netfpga: Reusable router architecture for experimental research. In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, page 1–7, New York, NY, USA, 2008. Association for Computing Machinery.

[14] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an openflow switch on the netfpga platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, page 1–9, New York, NY, USA, 2008. Association for Computing Machinery.

[15] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, page 127–132, New York, NY, USA, 2013. Association for Computing Machinery.

[16] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 99–110, New York, NY, USA, 2013. Association for Computing Machinery.

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[18] Yikai Lin, Ulaş C Kozat, John Kaippallimalil, Mehrdad Moradi, Anthony CK Soong, and Z Morley Mao. Pausing and resuming network flows using programmable buffers. In *Proceedings of the Symposium on SDN Research*, SOSR '18, page 7. ACM, 2018.

[19] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 357–371. ACM, 2018.

[20] M. Moradi, Y. Lin, Z. M. Mao, S. Sen, and O. Spatscheck. Softbox: A customizable, low-latency, and scalable 5g core network architecture. *IEEE Journal on Selected Areas in Communications*, 36(3):438–456, 2018.

[21] F. Tao, H. Zhang, A. Liu, and A. Y. C. Nee. Digital twin in industry: State-of-the-art. *IEEE Transactions on Industrial Informatics*, 15(4):2405–2415, 2019.

[22] J. Moyne, Y. Qamsane, E. C. Balta, I. Kovalenko, J. Faris, K. Barton, and D. M. Tilbury. A requirements driven digital twin framework: Specification and opportunities. *IEEE Access*, 8:107781–107801, 2020.

[23] J. Liu, Y. Li, M. Chen, W. Dong, and D. Jin. Software-defined internet of things for smart urban sensing. *IEEE Communications Magazine*, 53(9):55–63, 2015.

[24] K. Liu, J. K. Y. Ng, V. C. S. Lee, S. H. Son, and I. Stojmenovic. Cooperative data scheduling in hybrid vehicular ad hoc networks: Vanet as a software defined network. *IEEE/ACM Transactions on Networking*, 24(3):1759–1773, 2016.

[25] A. Jindal, G. S. Aujla, N. Kumar, R. Chaudhary, M. S. Obaidat, and I. You. Sedative: Sdn-enabled deep learning architecture for network traffic control in vehicular cyber-physical systems. *IEEE Network*, 32(6):66–73, 2018.

[26] Felipe Lopez, Yuru Shao, Z. Morley Mao, James Moyne, Kira Barton, and Dawn Tilbury. A software-defined framework for the integrated management of smart manufacturing systems. *Manufacturing Letters*, 15:18 – 21, 2018.

[27] Efe C Balta, Dawn M Tilbury, and Kira Barton. A centralized framework for system-level control and management of additive manufacturing fleets. In *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, pages 1071–1078. IEEE, 2018.

[28] Yassine Qamsane, Chien-Ying Chen, Efe C Balta, Bin-Chou Kao, Sibin Mohan, James Moyne, Dawn Tilbury, and Kira Barton. A unified digital twin framework for

real-time monitoring and evaluation of smart manufacturing systems. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, pages 1394–1401. IEEE, 2019.

[29] Ryu sdn framework. https://osrg.github.io/ryu/. (Accessed on 08/09/2020).

[30] Floodlight OpenFlow Controller. http://www.projectfloodlight.org/floodlight/. (Accessed on 08/09/2020).

[31] P4. https://p4.org/. (Accessed on 08/09/2020).

[32] p4lang/p4runtime: Specification documents for the p4runtime control-plane api. https://github.com/p4lang/p4runtime. (Accessed on 08/09/2020).

[33] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6. IEEE, 2014.

[34] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[35] Yikai Lin, Ajay Mahimkar, Bo Han, Zihui Ge, Vijay Gopalakrishnan, and Z Morley Mao. Egret: Simplifying traffic management for physical and virtual network functions. In *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '19, page 7. ACM, 2019.

[36] Yikai Lin, Yuru Shao, Xiao Zhu, Junpeng Guo, Kira Barton, and Z Morley Mao. Add: Application and data-driven controller design. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, pages 84–90. ACM, 2019.

[37] Docker. https://www.docker.com/. (Accessed on 08/09/2020).

[38] Open vSwitch. http://www.openvswitch.org/. (Accessed on 08/09/2020).

[39] Jeffrey G Andrews, Stefano Buzzi, Wan Choi, Stephen V Hanly, Angel Lozano, Anthony CK Soong, and Jianzhong Charlie Zhang. What will 5G be? *IEEE Journal on selected areas in communications*, 32(6):1065–1082, 2014.

[40] 5G Vision Brochure. https://5g-ppp.eu/wp-content/uploads/2015/02/5G-Vision-Brochure-v1.pdf, 02 2015.

[41] AT&T. AT&T Unveils 5G Roadmap Including Trials In 2016. http://about.att.com/story/unveils_5g_roadmap_including_trials.html, 02 2016.

[42] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 163–174, New York, NY, USA, 2014. Association for Computing Machinery.

[43] V. Yazici, U. C. Kozat, and M. O. Sunay. A new control plane for 5G network architecture with a case study on unified handoff, mobility, and routing management. *IEEE Communications Magazine*, 52(11):76–85, Nov 2014.

[44] NGMN Alliance. 5G White Paper. *Next Generation Mobile Networks, White Paper*, 2015.

[45] Service-Oriented 5G Core Networks. `http://carrier.huawei.com/~/media/CNBG/Downloads/track/HeavyReadingWhitepaperServiceOriented5GCoreNetworks.pdf`. (Accessed on 08/09/2020).

[46] OpenDaylight. `https://www.opendaylight.org/`. (Accessed on 08/09/2020).

[47] ONOS - A new carrier-grade SDN network operating system designed for high availability, performance, scale-out. `https://onosproject.org/`. (Accessed on 08/09/2020).

[48] Aaron Gember-Jacobson and Aditya Akella. Improving the safety, scalability, and efficiency of network function state transfers. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, pages 43–48. ACM, 2015.

[49] Wonil Roh, Ji-Yun Seol, Jeongho Park, Byunghwan Lee, Jaekon Lee, Yungsoo Kim, Jaeweon Cho, Kyungwhoon Cheun, and Farshid Aryanfar. Millimeter-wave beamforming as an enabling technology for 5G cellular communications: Theoretical feasibility and prototype results. *IEEE Communications Magazine*, 52(2):106–113, 2014.

[50] Sanjib Sur, Vignesh Venkateswaran, Xinyu Zhang, and Parmesh Ramanathan. 60 ghz indoor networking through flexible beams: A link-level profiling. *SIGMETRICS Perform. Eval. Rev.*, 43(1):71–84, June 2015.

[51] Roger Piqueras Jover and Ilona Murynets. Connection-less communication of IoT devices over LTE mobile networks. In *Proc. of IEEE SECON'15*, pages 247–255. IEEE, 2015.

[52] gRPC. `https://grpc.io/`. (Accessed on 08/09/2020).

[53] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. mSwitch: a highly-scalable, modular software switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, page 1. ACM, 2015.

[54] TUN/TAP - Wikipedia. https://en.wikipedia.org/wiki/TUN/TAP. (Accessed on 08/09/2020).

[55] netmap. http://info.iet.unipi.it/~luigi/netmap/. (Accessed on 08/09/2020).

[56] 5G: A technology vision. https://www.huawei.com/mediafiles/CORPORATE/PDF/Magazine/WinWin/HW_329327.pdf, 03 2014.

[57] 3GPP TR 138 913 V14.2.0. Study on Scenarios and Requirements for Next Generation Acess Technologies, May 2017.

[58] Internet Connection Speed Recommendations. https://help.netflix.com/en/node/306. (Accessed on 08/09/2020).

[59] Donghyuk Han, Sungjin Shin, Hyoungjun Cho, Jong-Moon Chung, Dongseok Ok, and Iksoon Hwang. Measurement and stochastic modeling of handover delay and interruption time of smartphone real-time applications on LTE networks. *IEEE Communications Magazine*, 53(3):173–181, 2015.

[60] Yin Xu, Zixiao Wang, Wai Kay Leong, and Ben Leong. An end-to-end measurement study of modern cellular data networks. In *International Conference on Passive and Active Network Measurement*, pages 34–45. Springer, 2014.

[61] SDN / OpenFlow / Message Layer / FlowMod — Flowgrammable. http://flowgrammable.org/sdn/openflow/message-layer/flowmod/. (Accessed on 08/09/2020).

[62] Mehrdad Moradi, Wenfei Wu, Li Erran Li, and Zhuoqing Morley Mao. Softmow: Recursive and reconfigurable cellular wan architecture. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, CoNEXT '14, pages 377–390. ACM, 2014.

[63] Xin Jin, Li Erran Li, Laurent Vanbever, and Jennifer Rexford. Softcell: Scalable and flexible cellular core network architecture. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, CoNEXT '13, pages 163–174. ACM, 2013.

[64] Binh Nguyen, Arijit Banerjee, Vijay Gopalakrishnan, Sneha Kasera, Seungjoon Lee, Aman Shaikh, and Jacobus Van der Merwe. Towards understanding TCP performance on LTE/EPC mobile networks. In *Proceedings of the 4th workshop on All things cellular: operations, applications, & challenges*, pages 41–46. ACM, 2014.

[65] Ayman Elnashar and Mohamed A El-Saidny. Looking at LTE in practice: A performance analysis of the LTE system based on field test results. *IEEE Vehicular Technology Magazine*, 8(3):81–92, 2013.

[66] Tofino switch — barefoot. https://barefootnetworks.com/products/brief-tofino/. (Accessed on 08/09/2020).

[67] Marek Irland. Buffer management in a packet switch. *IEEE transactions on Communications*, 26(3):328–337, 1978.

[68] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, pages 44–57. ACM, 2016.

[69] K. Kogan, D. Menikkumbura, G. Petri, Y. Noh, S. Nikolenko, A. Sirotkin, and P. Eugster. A programmable buffer management platform. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.

[70] Mina Tahmasbi Arashloo, Monia Ghobadi, Jennifer Rexford, and David Walker. HotCocoa: Hardware Congestion Control Abstractions. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 108–114. ACM, 2017.

[71] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[72] James S Plank, Micah Beck, Wael R Elwasif, Terence Moore, Martin Swany, and Rich Wolski. The internet backplane protocol: Storage in the network. In *In Proceedings of the Network Storage Symposium*. Citeseer, 1999.

[73] Yang Wang, Gaogang Xie, Zhenyu Li, Peng He, and Kavé Salamatian. Transparent flow migration for nfv. In *Network Protocols (ICNP), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[74] Arijit Banerjee et al. Scaling the LTE Control-Plane for Future Mobile Access. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2015.

[75] Zafar Ayyub Qazi et al. KLEIN: A Minimally Disruptive Design for an Elastic Cellular Core. In *Proceedings of the 2nd ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '16, 2016.

[76] M-CORD Open Source Reference Solution for 5G Mobile Wireless Networks. `https://www.opennetworking.org/solutions/m-cord/`. (Accessed on 08/09/2020).

[77] Ivan Seskar, Kiran Nagaraja, Sam Nelson, and Dipankar Raychaudhuri. Mobilityfirst future internet architecture project. In *Proceedings of the 7th Asian Internet Engineering Conference*, pages 1–3. ACM, 2011.

[78] Mehrdad Moradi, Feng Qian, Qiang Xu, Zhuoqing Morley Mao, Darrell Bethea, and Michael K Reiter. Caesar: High-speed and memory-efficient forwarding engine for future Internet architecture. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems*, ANCS '15, pages 171–182. IEEE Computer Society, 2015.

[79] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, et al. Maelstrom: mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 373–389, 2018.

[80] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. Zupdate: Updating data center networks with zero loss. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 411–422, New York, NY, USA, 2013. Association for Computing Machinery.

[81] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 539–550, New York, NY, USA, 2014. Association for Computing Machinery.

[82] Openconfig — vendor-neutral, model-driven network management designed by users. `http://www.openconfig.net/`. (Accessed on 08/09/2020).

[83] Cisco. Managing configuration files configuration guide, cisco ios xe release 3s - configuration replace and configuration rollback [cisco ios xe 3s] - cisco. `https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/config-mgmt/configuration/xe-3s/config-mgmt-xe-3s-book/cm-config-rollback.html`, 2019.

[84] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. Risk based planning of network changes in evolving data centers. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 414–429, New York, NY, USA, 2019. Association for Computing Machinery.

[85] Mininet: An instant virtual network on your laptop (or other pc). `http://mininet.org/`. (Accessed on 08/09/2020).

[86] Quagga. `https://www.quagga.net/docs/quagga.html`. (Accessed on 08/09/2020).

[87] Nginx — http load balancing. `https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/`. (Accessed on 08/09/2020).

[88] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, New York, NY, USA, 2008. Association for Computing Machinery.

[89] Alex X Liu, Chad R Meiners, and Eric Torng. Tcam razor: A systematic approach towards minimizing packet classifiers in tcams. *IEEE/ACM Transactions on Networking (TON)*, 18(2):490–500, 2010.

[90] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. vcrib: Virtualized rule management in the cloud. In *Presented as part of the*, 2012.

[91] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable rule management for data centers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, page 157–170, USA, 2013. USENIX Association.

[92] Martin Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the network forwarding plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, page 8. ACM, 2010.

[93] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the one big switch abstraction in software-defined networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, CoNEXT'13, pages 13–24. ACM, 2013.

[94] Chen Sun, Jun Bi, Zili Meng, Xiao Zhang, and Hongxin Hu. Ofm: Optimized flow migration for nfv elasticity control. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2018.

[95] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[96] Alaitz Mendiola, Jasone Astorga, Eduardo Jacob, and Marivi Higuero. A survey on the contributions of software-defined networking to traffic engineering. *IEEE Communications Surveys & Tutorials*, 19(2):918–953, 2016.

[97] David Ke Hong, Yadi Ma, Sujata Banerjee, and Z Morley Mao. Incremental deployment of sdn in hybrid enterprise and isp networks. In *Proceedings of the Symposium on SDN Research*, page 1. ACM, 2016.

[98] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 207–218, New York, NY, USA, 2013. Association for Computing Machinery.

[99] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 27–38, New York, NY, USA, 2014. Association for Computing Machinery.

[100] Rohan Gandhi, Y. Charlie Hu, Cheng kok Koh, Hongqiang (Harry) Liu, and Ming Zhang. Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 473–485, Santa Clara, CA, July 2015. USENIX Association.

[101] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 523–535, USA, 2016. USENIX Association.

[102] Istio — traffic shifting. `https://istio.io/docs/tasks/traffic-management/traffic-shifting/`, 2019.

[103] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 15–28. ACM, 2017.

[104] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, volume 18, pages 125–139, 2018.

[105] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. Balancing on the edge: Transport affinity without network state. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), Renton, WA*, 2018.

[106] Junaid Khalid, Mark Coatsworth, Aaron Gember-Jacobson, and Aditya Akella. A standardized southbound api for vnf management. In *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization*, pages 38–43. ACM, 2016.

[107] Netflow. `https://www.solarwinds.com/what-is-netflow`. (Accessed on 08/09/2020).

[108] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *Proceedings of the Third ACM/IEEE Symposium on Edge Computing*, pages 115–131. IEEE, 2018.

[109] Yousra Alkabani and Farinaz Koushanfar. Active hardware metering for intellectual property protection and security. In *USENIX security symposium*, pages 291–306, 2007.

[110] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 421–434, New York, NY, USA, 2015. Association for Computing Machinery.

[111] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. Wide-area analytics with multiple resources. In *Proceedings of the Thirteenth EuroSys Conference*, page 12. ACM, 2018.

[112] Xianghang Mi, Feng Qian, and Xiaofeng Wang. Smig: Stream migration extension for http/2. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, CONEXT'16, pages 121–128. ACM, 2016.

[113] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, page 289–300, New York, NY, USA, 2005. Association for Computing Machinery.

[114] The most popular database for modern apps — mongodb. `https://www.mongodb.com/`. (Accessed on 08/09/2020).

[115] Opc foundation. `https://opcfoundation.org/`. (Accessed on 08/09/2020).

[116] Protocol buffers. `https://developers.google.com/protocol-buffers/`. (Accessed on 08/09/2020).

[117] Graphql. `https://graphql.org/`. (Accessed on 08/09/2020).

[118] Manufacturing software solutions, rockwell factorytalk. `https://www.rockwellautomation.com/rockwellsoftware/products/overview.page`, 2018.

[119] Laizhong Cui, F Richard Yu, and Qiao Yan. When big data meets software-defined networking: Sdn for big data and big data for sdn. *IEEE network*, 30(1):58–65, 2016.

[120] Albert Mestres, Alberto Rodriguez-Natal, Josep Carner, Pere Barlet-Ros, Eduard Alarcón, Marc Solé, Victor Muntés-Mulero, David Meyer, Sharon Barkai, Mike J Hibbett, et al. Knowledge-defined networking. *ACM SIGCOMM Computer Communication Review*, 47(3):2–10, 2017.

[121] Alexander Clemm, Mouli Chandramouli, Nitish Gupta, Robert Lerche, Ashwin Pankaj, Manjunath Patil, Ganesan Rajam, V Anbalagan, Joe Zhang, and Yifan Zhang. Dna: An sdn framework for distributed network analytics (demo paper). In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 1143–1144. IEEE, 2015.

[122] Haojun Huang, Hao Yin, Geyong Min, Hongbo Jiang, Junbao Zhang, and Yulei Wu. Data-driven information plane in software-defined networking. *IEEE Communications Magazine*, 55(6):218–224, 2017.

[123] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM'17, pages 85–98. ACM, 2017.

[124] Pradeeban Kathiravelu, Peter Van Roy, and Luís Veiga. Sd-cps: software-defined cyber-physical systems. taming the challenges of cps with workflows at the edge. *Cluster Computing*, 22(3):661–677, 2019.

[125] BBC. Coronavirus: Can we 3d-print our way out of the ppe shortage? `https://www.bbc.com/news/health-52201696`, April 2020.

[126] openconfig/gnmi: grpc network management interface. `https://github.com/openconfig/gnmi`. (Accessed on 08/09/2020).

[127] Redis cluster specification. `https://redis.io/topics/cluster-spec`. (Accessed on 08/09/2020).

[128] Replication — mongodb manual. `https://docs.mongodb.com/manual/replication/`. (Accessed on 08/09/2020).

[129] Apache kafka. `https://kafka.apache.org/`. (Accessed on 08/09/2020).

[130] Pub/sub — redis. `https://redis.io/topics/pubsub`. (Accessed on 08/09/2020).

[131] Apache hadoop. `https://hadoop.apache.org/`. (Accessed on 08/09/2020).

[132] Apache spark™ — unified analytics engine for big data. `https://spark.apache.org/`. (Accessed on 08/09/2020).

[133] Apache hadoop nextgen mapreduce (yarn). `https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html`. (Accessed on 08/09/2020).

[134] Redis. `https://redis.io/`. (Accessed on 08/09/2020).

[135] Pusher — leader in realtime technologies. `https://pusher.com/`. (Accessed on 08/09/2020).

[136] The reactive extensions for python (rxpy). `https://rxpy.readthedocs.io/en/latest/`. (Accessed on 08/09/2020).

[137] Reactivex. `http://reactivex.io/`. (Accessed on 08/09/2020).

[138] Using pipelining to speedup redis queries. https://redis.io/topics/pipelining. (Accessed on 08/09/2020).

[139] Global interpreter lock. https://en.wikipedia.org/wiki/Global_interpreter_lock. (Accessed on 08/09/2020).

[140] Database index. https://en.wikipedia.org/wiki/Database_index. (Accessed on 08/09/2020).

[141] The Washington Post. Ford and gm are racing to build coronavirus ventilators, but their efforts may be too late. https://www.washingtonpost.com/business/2020/04/04/ventilators-coronavirus-ford-gm/, April 2020.

[142] Professional 3d printing made accessible — ultimaker. https://ultimaker.com/. (Accessed on 08/09/2020).

[143] Efe C Balta, Dawn M Tilbury, and Kira Barton. A digital twin framework for performance monitoring and anomaly detection in fused deposition modeling. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, pages 823–829. IEEE, 2019.

[144] P4-16 language specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html. (Accessed on 08/09/2020).

[145] Syn flood. https://en.wikipedia.org/wiki/SYN_flood. (Accessed on 08/09/2020).

[146] David Moore, Colleen Shannon, Douglas J Brown, Geoffrey M Voelker, and Stefan Savage. Inferring internet denial-of-service activity. *ACM Transactions on Computer Systems (TOCS)*, 24(2):115–139, 2006.

[147] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.

[148] Nasrin Sultana, Naveen Chilamkurti, Wei Peng, and Rabei Alhadad. Survey on sdn based network intrusion detection system using machine learning approaches. *Peer-to-Peer Networking and Applications*, 12(2):493–501, 2019.

[149] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *NDSS*. The Internet Society, 2018.

[150] p4lang/behavioral-model: The reference p4 software switch. https://github.com/p4lang/behavioral-model. (Accessed on 08/09/2020).

[151] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC, August 2017. USENIX Association.

[152] grpc — a high-performance, open source universal rpc framework. `https://grpc.io/`. (Accessed on 08/09/2020).

[153] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 182–196, New York, NY, USA, 2013. Association for Computing Machinery.

[154] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Rao Kompella. Elasticon; an elastic distributed sdn controller. In *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 17–27. IEEE, 2014.

[155] Pingping Lin, Jun Bi, Stephen Wolff, Yangyang Wang, Anmin Xu, Ze Chen, Hongyu Hu, and Yikai Lin. A west-east bridge based sdn inter-domain testbed. *IEEE Communications Magazine*, 53(2):190–197, 2015.